Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Authenticated Payload Encryption Scheme for Internet of Things Systems over the MQTT Protocol

Sorcha Nolan
13317836

May 17, 2018

A dissertation submitted in partial fulfilment
of the requirements for the degree of
MAI (Computer & Electronic Engineering)

Declaration

Permission to Lend and/or Copy

Acknowledgments

I would like to thank a number of people who have made this dissertation possible. First of all, I would like to sincerely thank my supervisor Dr. Stefan Weber for all of his help, guidance and valued time given throughout the project. The constant support, consultation and encouragement have been invaluable in the completion of the following dissertation.

I am very grateful for the technical advice I have received from Dr. Jonathan Dukes, along with his introduction to the Internet of Things, providing me with the strong background necessary to complete this dissertation.

I would also like to include a special note of thanks to all those involved with the Masters programme at Trinity College Dublin, in both the Engineering department and Computer Science department. In particular, Dr Mike Brady, MAI coordinator in the Department of Computer Science.

Finally, I would like to express my sincere gratitude to my family for their advice and endless encouragement in my academic endeavours, without them this dissertation would certainly not have been possible.

Sorcha Nolan

University of Dublin, Trinity College
May 2018

Abstract

The Internet of Things (IoT) comprises resource-constrained devices connected to the Internet, interacting with the real world within a wide range of applications. The recent large-scale commercialisation of these low-powered devices has lead to significant security concerns. Message Queue Telemetry Transport (MQTT) is the most widely used application-layer protocol over IoT. A robust and lightweight security scheme is required for use with this protocol, as the security aspect has been omitted from the protocol design. Transport Layer Security (TLS) is recommended in this scenario, though it is often unsuitable due to its resource-intensive nature and lack of end-to-end security provision. A scheme has been proposed using symmetric-key payload encryption, designed entirely over the MQTT protocol. This solution requires minimal overhead on the IoT device, offloading the bulk of the resource-intensive computation onto a central Key Management Service. Overhead is minimised with regard to bandwidth, time-to-idle, memory and computation, and improves upon TLS in all of these areas. The scheme successfully provides secure and authenticated end-to-end communication between clients in the system.

Contents

List of Tables

List of Figures

Nomenclature

6LBR	6LoWPAN Border Router
6LN	6LoWPAN Link Node
6LoWPAN	IPv6 over Low-Power Wireless Personal Area Networks
ABE	Attribute-Based Encryption
ACL	Access Control List
AES	Advanced Encryption Standard
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
BLE	Bluetooth Low Energy
CA	Certificate Authority
CBC-MAC	Cipher Block-Chaining Message Authentication Code
CCM	Counter Mode with CBC-MAC
CIA	Confidentiality, Integrity & Availability
CoAP	Constrained Application Protocol
DDoS	Distributed Denial of Service
DES	Data Encryption Standard
DTLS	Datagram Transport Layer Security
DoS	Denial of Service
ECDSA	Elliptic Curve Digital Signature Algorithm
GCM	Galois/Counter Mode
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol (IPv4/IPv6)
IV	Initialisation Vector
IoT	Internet of Things
KMS	Key Management Service
LLN	Low-powered & Lossy Networks
LRU	Least Recently Used
LSB	Least Significant Bit
M2M	Machine-to-Machine
MAC	Message Authentication Code

MQTT	Message Queue Telemetry Transport
MQTT-SN	Message Queue Telemetry Transport
MSB	Most Significant Bit
MTU	Maximum Transmission Unit
NIST	National Institute of Standards and Technology
OSI	Open Systems Interconnection model
QoS	Quality of Service
REST	Representational State Transfer
RFID	Radio-Frequency Identification
RSA	RivestShamirAdleman cryptosystem
RTOS	Real-Time Operating System
SDK	Software Development Kit
SHA	Secure Hash Algorithm
SQL	Structured Query Language
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLRU	Time-aware Least Recently Used
TLS	Transport Layer Security
UDP	User Datagram Protocol
XMPP	Extensible Messaging and Presence Protocol

Chapter 1

Introduction

The Internet of Things (IoT) is connecting physical objects to the Internet at an unprecedented rate - from smart homes to smart cities, the healthcare system to autonomous vehicles. Gartner estimates that there will be 21 billion connected "things" by 2020 (Hoeppe, 2017). IoT has provided a platform to allow unprecedented development and discovery within a wide range of applications. Along with this potential, however, are the rapidly increasing requirements for the large-scale deployment and commercialisation of IoT, resulting in major security concerns (Alaba et al., 2017). This is due to the fact that implementing security is resource-intensive, and can be a difficult and expensive endeavour to apply to low-powered devices. It is therefore not always seen as a worthwhile pursuit.

The majority of IoT devices and networks are constrained in resources, depending on their application, design, location and topology. They can be constrained with regard to battery power, memory, processing power, bandwidth and transfer unit. For this reason, standard Internet transport protocols are not used for most IoT implementations. Lightweight protocols or adapted versions of standard protocols are instead used, allowing minimum processing and overhead on the IoT device. In order to keep these protocols as lightweight as possible, security protocols are generally omitted, allowing the implementer to decide the level of security necessary for their particular application (Nguyen et al., 2015).

One of the most prevalent application-layer protocols over IoT is Message Queue Telemetry Transport (MQTT). MQTT is a lightweight publish/subscribe messaging transport protocol for machine-to-machine (M2M) and IoT contexts. This employs a central broker, hosting topics that can be published and subscribed to by system clients. It is concerned only with message transmission, and it is the implementer's responsibility to provide appropriate security features (A. Banks, 2014). This is commonly achieved by using Transport Layer Security (TLS). While TLS may be the most straightforward security protocol to implement, it is not always the most appropriate one for an IoT application.

This chapter gives a brief introduction into the research area, context and contents of the dissertation to follow. The motivation and relevancy of the research area is delineated in Section 1.1. Section 1.2 gives an overview of the solution that has been proposed. Finally Section 1.3 outlines the structure of the dissertation, introducing each chapter and its respective contents.

1.1 Motivation

Security is a significant issue in resource-constrained IoT environments. The number of commercial IoT systems deployed without adequate security mechanisms is growing exponentially, leading to an abundance of vulnerabilities and insecurities. A study carried out by HP Fortify (2014) states that 70% of IoT devices currently in market do not have an adequate security implementation. The Mirai botnet attack (Jerkins, 2017) is a notable example of IoT security failure, where thousands of insecure IoT devices were infected by malware and controlled for use in a massive Distributed Denial of Service (DDoS) attack. These insecurities have lead to a lack of trust in IoT in some spheres, somewhat limiting confident growth in the industry. Security is difficult, particularly in a resource-constrained environment. It is imperative that research in security keeps up with the fast-paced ongoing developments in other aspects of IoT.

1.2 Overview

This dissertation presents a payload encryption scheme for use in MQTT-based IoT systems. The proposed scheme runs entirely over the MQTT protocol, to be integrated seamlessly into an IoT application. The scheme aims to provide a flexible, robust and thorough security architecture. Capabilities are provided for the clients in the system to encrypt and decrypt messages using symmetric-key encryption for confidentiality, as well as ensuring the integrity of the message through Message Authentication Codes (MACs). These messages remain encrypted while being stored by the broker, ensuring end-to-end security within the publish/subscribe framework. A central Key Management Service (KMS) provides key exchange, key lifecycle management, authentication, access control and monitoring services to the system. This service is unconstrained in resources and handles any complex or extraneous processing, allowing only the essential computation to occur on the IoT devices.

1.3 Roadmap

The structure of the dissertation is laid out as follows. Chapter 2 contains the State of the Art, describing the background, existing work and research going into the area of IoT security. This details IoT protocols, particularly MQTT, security implementations and considerations and closely-related research. These are examined to formulate the problem to be solved. Chapter 3 is the Design chapter, specifying how and why the solution was devised in light of the problem formulation. An in-depth exposition of the scheme implementation is presented in Chapter 4. This outlines the deployment architecture, technologies used and components involved in the development of the system. An evaluation of the system is carried out within Chapter 5, providing an analysis of overhead, security and a general discussion of design decisions and scheme implementation. Finally, conclusions and future work are considered within Chapter 6.

Chapter 2

State of the Art

The State of the Art describes the background, existing work and current research going into the area of IoT security. Section 2.1 begins with an analysis of IoT protocols and protcols providing for a resource-constrained environment. This section focuses on MQTT, as well as detailing other relevant protocols such CoAP, 6LoWPAN and Bluetooth Low Energy. Following this, Section 2.2 gives an in-depth review of security, in general and within the scope of IoT. Within this section exists an assessment of various security implementations, with a more granular focus on the most relevant implementations for the solution design. These include TLS, payload encryption and symmetric-key encryption. Section 2.3 reviews current research into closely-related areas, providing a critical view of this research, used to inform the subsequent design. These sections are all used to consolidate a problem formulation, which is detailed in Section 2.4.

2.1 Internet of Things Protocols

The Internet of Things is a framework that allows physical objects to be connected to the Internet, share information and coordinate decisions. Due to the heterogeneous and varied nature of IoT, a suite of IoT protocols have been developed to satisfy the unique concerns that arise. A flexible, multi-layered architecture is necessary to accommodate the needs of billions of heterogeneous devices (Al-Fuqaha et al., 2015). Standard Internet protocols have been designed over the years for a human-to-machine, resource-unconstrained environment, with little concern for added communication overhead. On the other hand, the IoT environment involves mainly M2M communication, with several small, low-powered resource-constrained devices. Protocols have been developed to minimise processing on the IoT device, and be able to integrate with standard Internet protocols to form an Internet connection.

A universal IoT architecture has yet to be established, though several potential reference models have been proposed with varying levels of complexity (Mosenia & Jha, 2017). A three-layer architecture (application, network and perception layers) and a five-layer, middleware-based architecture have both been proposed. In 2014, CISCO developed a seven-layer architecture that has the potential for standardisation across IoT (Fig. 2.1).

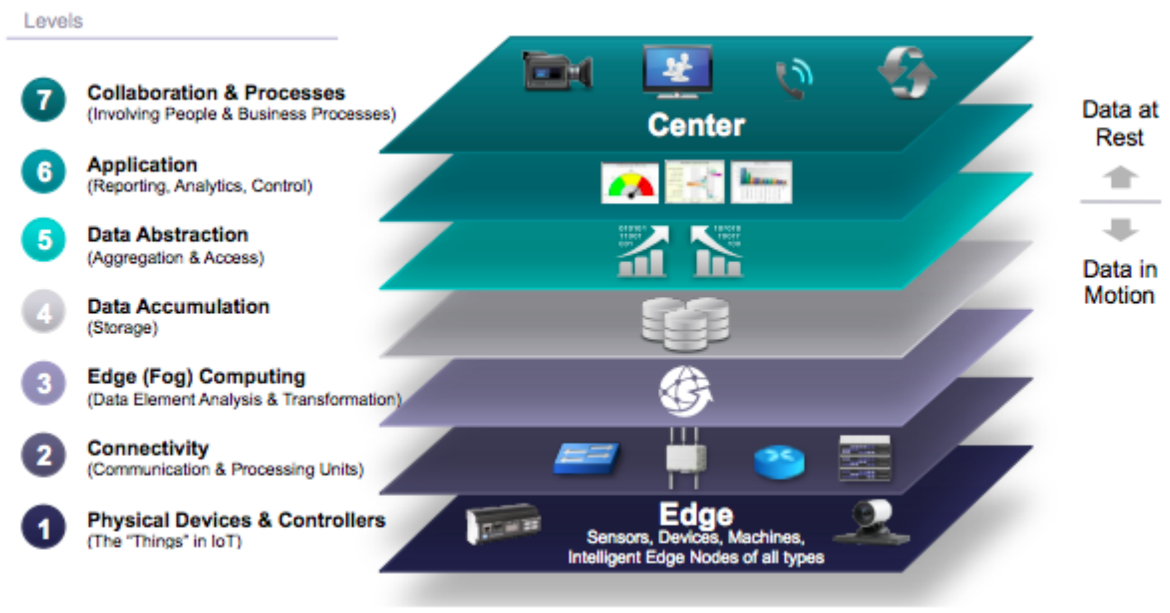
Levels
7 Collaboration & Processes
(Involving People & Business Processes)
6 Application
(Reporting, Analytics, Control)
5 Data Abstraction
(Aggregation & Access)
4 Data Accumulation
(Storage)
3 Edge (Fog) Computing
(Data Element Analysis & Transformation)
2 Connectivity
(Communication & Processing Units)
1 Physical Devices & Controllers
(The "Things" in IoT)
Center
Edge
Sensors, Devices, Machines,
Intelligent Edge Nodes of all types
Data at Rest
Data in Motion

Figure 2.1: CISCO Seven-Level Reference Model (CISCO, 2014).

The first level consists of the physical devices - such as sensors, actuators and RFID tags. The next level is the communication layer, consisting of components and protocols that enable transmission of information between devices and to the cloud. Level 3 encapsulates edge computing, enabling data to be processed close to or at the source. This process reduces computational load and decreases latency (Morabito et al., 2018), and is particularly useful in real-time applications. Level 4 is the data accumulation layer, where network packets are filtered according to storage requirements and converted to database tables. The data abstraction layer renders the data for more efficient processing, using processes such as normalisation, consolidation and indexing. The application layer is where the data is interpreted and analysed. The final layer represents the users and business processes that make use of the data and system as a whole.

Several communication protocols have been developed or adapted to suit the IoT paradigm, and serve the needs of layer 2 of the CISCO reference model. In order to adequately serve the resource-constrained devices prevalent in IoT - compressed, lightweight and efficient protocols are necessary. Protocols must be robust and self-healing to serve M2M communication, often over low-powered and lossy networks (LLNs). The protocols should support the transmission of short bursts of information with a small number of bytes, before allowing the device to return to standby mode. Ideally, communication should be connectionless, asynchronous and event-driven (Karagiannis et al., 2015).

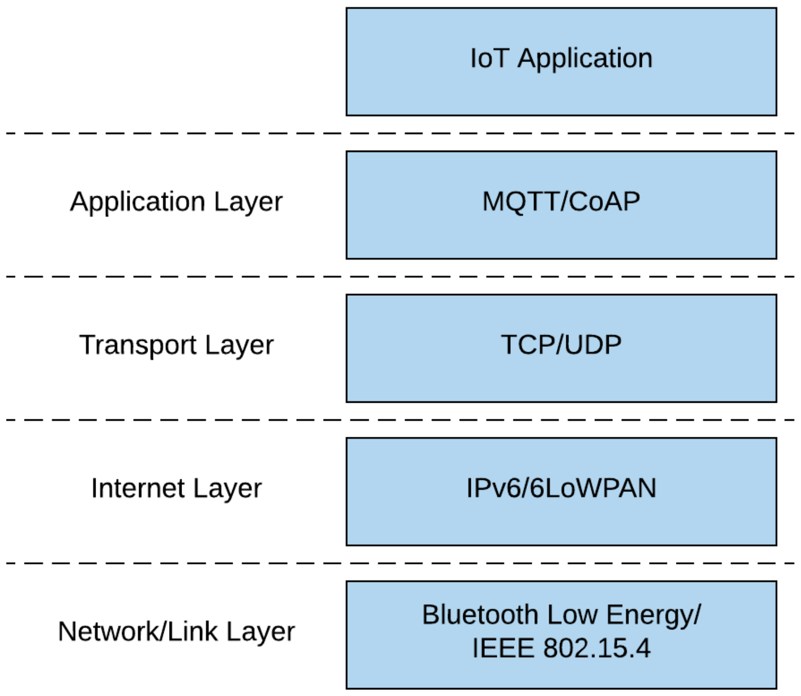

Figure 2.2: IoT Communication Protocol Stack.

Fig. 2.2 depicts the IoT communication protocol stack, identifying common protocols at each layer that have been designed or adapted to fit the IoT model. The lower OSI layers are provided by network infrastructure protocols such as Bluetooth Low Energy (BLE) or IEEE 802.15.4. The Internet layer creates IPv6 packets from IoT data to transmit. 6LoWPAN (IPv6 over Low-Power Wireless Personal Area Networks) is a standard that allows adaptation between constrained IoT protocols and IPv6. Several application-layer IoT protocols exist, the most prevalent being MQTT and CoAP. MQTT runs over TCP as the transport layer, and CoAP uses UDP. The IoT application interacts with the application-layer messages to utilise and act upon the information transmitted.

A common IoT network configuration can be seen in Fig. 2.3. This consists of a number of resource-constrained devices in an environment, communicating with an IoT gateway that forwards packets to the cloud. This gateway can also have edge computing capabilities, as in Fig. 2.1 level 3. The gateway can provide support on all levels of the IoT protocol stack. Network standards such as BLE have a low transmission range (approximately 100m) and a gateway is necessary to act as the central router for any BLE

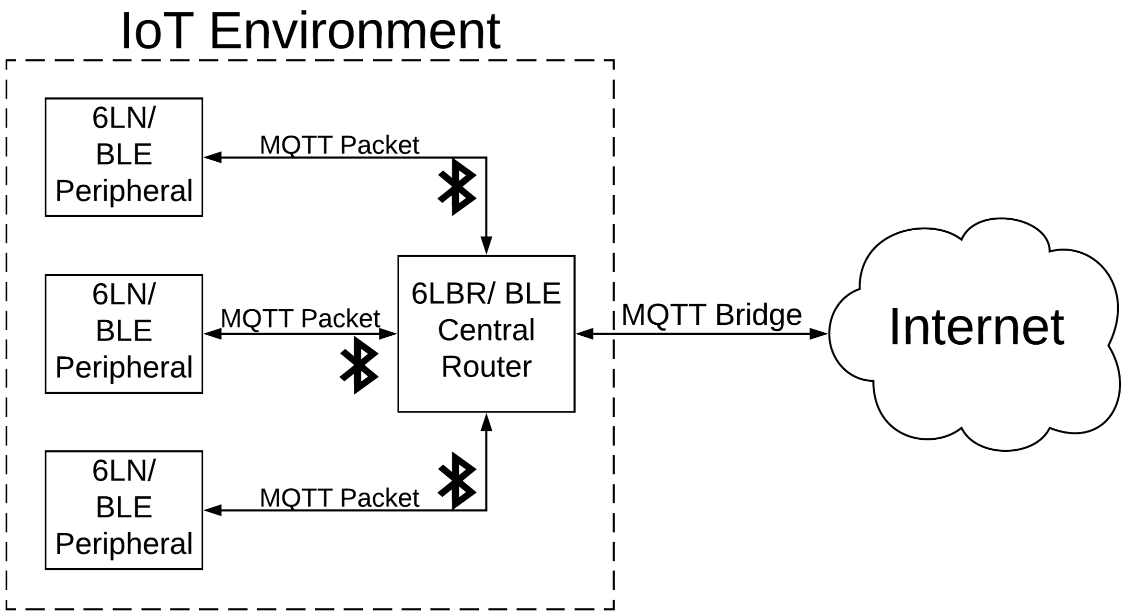

Figure 2.3: Border router setup.

peripherals. 6LoWPAN also requires a border router (6LBR) that can convert 6LoWPAN packets from link nodes (6LNs) to regular IPv6 packets. On the application layer, MQTT can use the IoT gateway as a broker bridge, which forwards all packets to an identical broker hosted in an accessible location such as on the cloud.

2.1.1 Message Queue Telemetry Transport

MQTT is the most widely used application-layer transport protocol over IoT. It is an extremely lightweight broker-based publish/subscribe messaging protocol, layered on top of TCP/IP, and often 6LoWPAN. It is designed for use over networks with low power and bandwidth, high latency and variable availability (Karagiannis et al., 2015). These aspects make MQTT an ideal protocol for use in IoT systems.

MQTT is made up of publishers, subscribers and a broker, as seen in Fig. 2.4. IoT devices such as sensors can publish their data regularly to a particular topic hosted by the MQTT broker. An IoT server can subscribe to this particular topic, allowing it to receive this information and process it as required. This process can be reversed to allow the server to invoke actuation on a device. This architecture allows asynchronous communication between publishers and subscribers. They have no direct connection and therefore do not need to be aware of the others' existence, which is a huge advantage of such a protocol (Ammar et al., 2018).

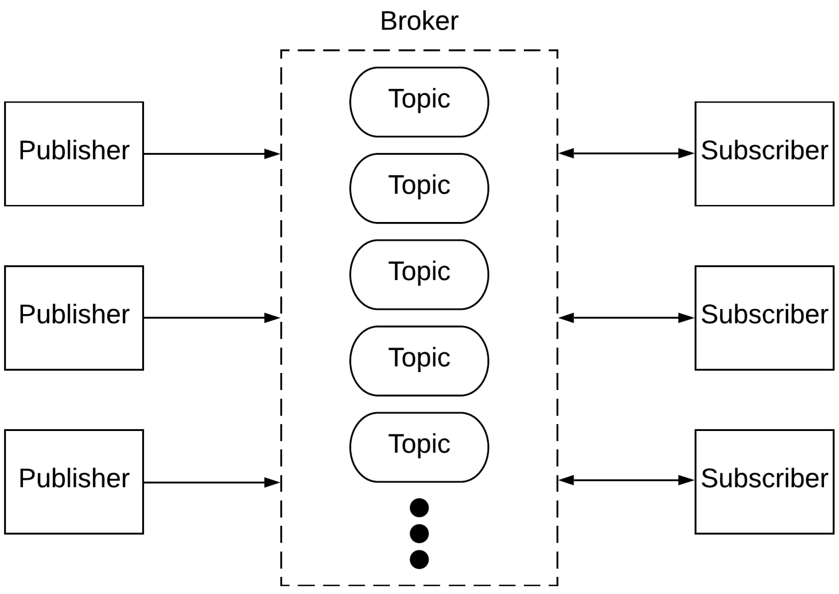

Figure 2.4: The architecture of MQTT.

The underlying transmission protocol, TCP/IP, aims to guarantee message delivery. This can be built upon by MQTT's quality of service (QoS) setting, which can be set to deliver a message *At Most Once*, *At Least Once* or *Exactly Once* (A. Banks, 2014). However, guaranteed delivery does not ensure data integrity, privacy or authenticity of the message. In order to ensure these considerations, a security layer must be implemented on top of the TCP/IP layer. To begin with, TCP/IP incurs a significant amount of transmission overhead compared with UDP. Transmission standards for low-powered wireless networks tend to use a small Maximum Transmission Unit (MTU). For example, IEEE 802.15.4 uses an MTU of 127 bytes, and the Bluetooth MTU can be as low as 27 bytes. This restricts the type and complexity of security protocols which can realistically be integrated into IoT protocols.

2.1.2 Constrained Application Protocol

The Constrained Application Protocol (CoAP), developed by the IETF, is a RESTful web transfer protocol based on HTTP for use in resource-constrained environments. It provides a simpler and more lightweight HTTP implementation, communicating through GET, POST, PUT and DELETE messages. CoAP is bound to UDP by default, making it a suitable protocol for IoT scenarios due to UDP's minimal and connectionless design (Z. Shelby, 2014). It is designed for flexible communication with HTTP through a proxy. CoAP is a secure protocol since it is built on top of datagram transport layer security (DTLS) to guarantee integrity and confidentiality of exchanged messages (Al-Fuqaha et al., 2015).

2.1.3 6LoWPAN

6LoWPAN denotes IPv6 over Low-Power Wireless Personal Area Networks, a standard aiming to provide the Internet Protocol to low-powered devices with limited capabilities. IPv6 allows traffic to be routed across the Internet, providing unique IP addresses to devices for identification. Communication across the Internet then occurs through the transmission of IPv6 (and IPv4) packets, consisting of 320 bit headers and a maximum payload size of 64KB. This is extremely unsuitable for the majority of IoT devices, that are unable to support the resource-intensive nature of the protocol. IoT data rates are much smaller, and most protocols will support an MTU of 127 bytes or smaller.

6LoWPAN adapts the IPv6 protocol to allow even the smallest of devices to communicate over the Internet. It implements an adaptation layer between the Network and Data-Link Layers in order to provide the required adaptation capabilities. These include header and payload compression, fragmentation and direct IP addressing for nodes. The nodes communicate with a 6LoWPAN border router (6LBR) in order to convert 6LoWPAN packets to ordinary IPv6 packets, to be transmitted on the Internet (Wang & Mu, 2017).

2.1.4 Bluetooth Low Energy

Bluetooth Low energy (BLE) is a wireless personal area network technology that leverages the classic Bluetooth framework for a resource-constrained environment. This allows low-powered communications to occur between devices such as sensors, actuators and mobile devices. BLE provides considerably reduced power consumption and cost compared to classic Bluetooth, while maintaining a similar communication range of approximately 100m. It was originally designed using a star topology, with BLE peripherals communicating with a central router, but further iterations have also provided support for mesh topologies (Darroudi & Gomez, 2017).

2.2 Security

Security should be a major factor in the design and development of technical systems. The purpose of security is to prevent losses, both accidental and malicious, and different systems will have different security priorities. In order to identify these, it must be known what is being protected and against whom. From there the weak points of the system can be determined. The nature of technology means that things are constantly changing and improving, and security must reflect that in terms of flexibility. Changes can be constantly seen in performance, capability, cost and environment. However, flexibility must be balanced with complexity, as an overly-complex system is fatal for security. Contingency planning can lead to bloated systems that are difficult to maintain. This can lead to security flaws appearing that are harder to identify and analyse (Bellovin, 2016).

There are a wide range of security aspects to consider in the design of a secure system. The three most crucial considerations for information security are known as the CIA triad - confidentiality, integrity and availability. Confidentiality determines a set of rules that limits unauthorised access to the information, which can be seen as privacy. Encryption is a common means of ensuring confidentiality. Integrity involves ensuring the legitimacy of the data, by maintaining its consistency, accuracy, and trustworthiness over its entire life cycle. Finally, availability guarantees reliable access to the information by authorized users. The CIA triad addresses a large number of security concerns, though new threats are always emerging (Mosenia & Jha, 2017). A system must also try to address accountability, auditing, non-repudiation and privacy policies.

Implementing security in IoT incurs limitations that do not generally arise in conventional Internet security. IoT comprises a wide range of smart devices, with different uses, size, energy capacity and computation power. Transmission protocols are lightweight with minimal overhead, before being integrated with standard Internet protocols. Security must be adapted to fit this paradigm, with each application demanding varied requirements (Kouicem et al., 2018). No inherent security framework exists for IoT, and conventional frameworks, such as TLS, are frequently unsuitable. Security is often not well recognised in commercial IoT, due to this lack of framework and regulation. This leads to a whole host of security concerns, and the undermining of trust in IoT systems that are perceived as insecure.

In order to ensure secure IoT communication, vulnerabilities should be addressed in terms of possible attacks that may occur. Common IoT intrusions are discussed by Mosenia & Jha (2017). An eavesdropping attack occurs when unencrypted data is sniffed over communication links. This can often be used to garner information in order to design other tailored attacks. Side-channel attacks are serious and complex attacks against encryption. A node may leak critical information under normal operation, for example its EM signature, or the regularity and timing of message transmission. They are usually non-invasive and therefore undetectable. Minimising leakage as much as possible will mitigate against these kinds of attacks.

Injecting fraudulent packets into the system can provoke chaos for an insecure system. Denial of Service (DoS) attacks can be used to jam transmission by bombarding the system with unnecessary requests. It can also be done intermittently to lower the performance of time-critical tasks. Malicious packets can be inserted along with normal communications, existing packets can be manipulated in some way, or packets can be replicated repeatedly to overload the system. Keeping track of state and previous packet information can help to defend against these type of attacks. Similarly, attacks can be conducted by routing manipulation - to spoof, redirect, misdirect or drop packets. This can occur through routing loops, false error messages or malicious nodes.

2.2.1 Transport Layer Security

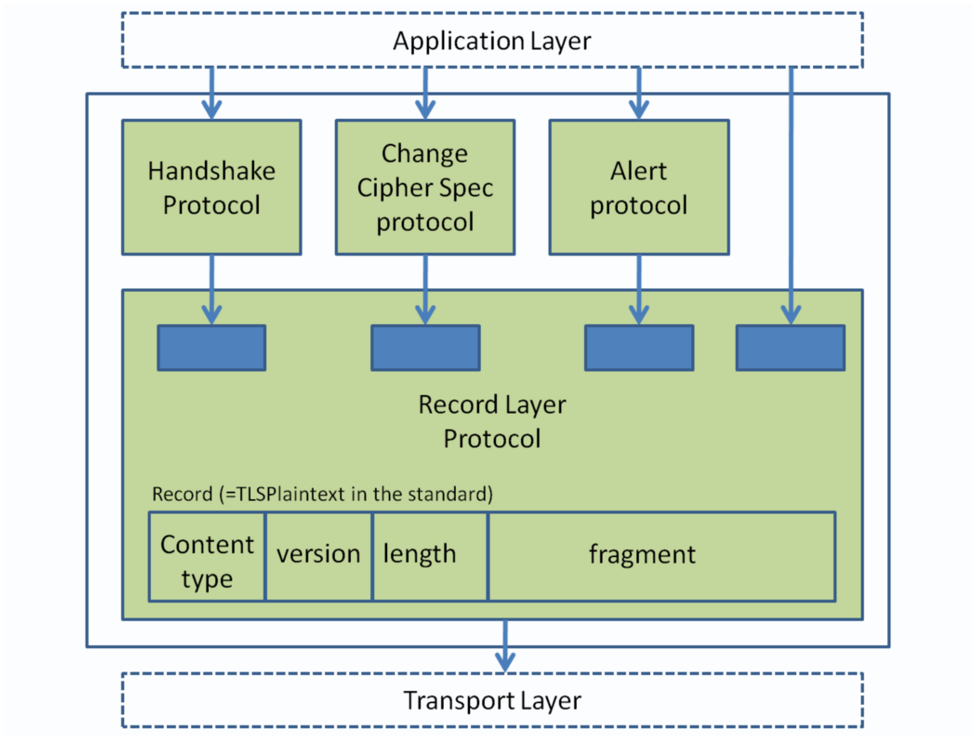

Figure 2.5: The architecture of TLS (Unger et al., 2012).

Transport Layer Security (TLS), also known as the Secure Socket Layer (SSL), is the de-facto standard for establishing a secure communication channel between two machines. As seen in Fig. 2.5, there are two facets to the protocol - the first establishes a connection between devices and negotiates parameters, and the second (Record Layer) carries the payload data, ensuring message integrity and confidentiality (T. Dierks, 2008). The handshake protocol that establishes a connection between devices uses asymmetric (public-key) cryptography to secure the interaction. A session-specific shared key is decided upon within this communication, which is then used to encrypt and decrypt the data transmitted between the two sources (symmetric encryption). TLS lies between the application layer (MQTT) and the transport layer (TCP/IP).

Due to the fact that TLS is a widespread and well-accepted existing protocol, new technologies often adopt it as their security implementation rather than designing a new protocol to suit the needs of the technology. In the case of resource-constrained IoT devices, there are many drawbacks to using TLS. In order for authentication to occur between devices, they must be able to handle asymmetric cryptography and X.509 certificates. These techniques require the devices to transmit a high amount of data and to run complex algorithms. Different authentication mechanisms that might be more feasible are not supported by TLS (Unger et al., 2012).

TLS incurs a significant amount of processing overhead when considering distributed topologies and multihop scenarios. There is no option to encrypt particular parts of a message in order for sensitive data to be protected while metadata remains accessible. Therefore in order to route the packet correctly, each node must decrypt and then re-encrypt it before passing it on (Unger et al., 2012). Another aspect to consider is that due to TLS existing on a lower layer, the payload is decrypted before it reaches the application layer. Encryption on an application level could prevent the caching of sensitive data in plaintext and foster secure data storage. With regard to MQTT, this means that data stored by the broker will be unencrypted, and anyone with access to the broker and topic could retrieve the unencrypted data.

2.2.2 Payload Encryption

The publish/subscribe paradigm is useful in IoT scenarios due to its event-based nature, decoupled communication, high throughput and scalability. However, these factors also give rise to confidentiality concerns. Both messages and subscriptions must be kept private, and the broker can be seen as a potential point of failure if it is compromised (Bacon et al., 2009). The broker may also be a third-party system that should not have access to sensitive information that passes through it. It must have access to routing information, however, such as the topic name. Ensuring confidentiality in such a system is therefore a compromise between the ability to accurately route publications and the risk of leaking information (Onica et al., 2016). Due to the decoupled nature of communication, end-to-end security cannot be ensured by using TLS. A secure channel can be established from the publisher to broker, and broker to subscriber, but the data stored in the broker will be unencrypted. The only way to adequately ensure end-to-end security in a publish/-subscribe system is to implement some form of payload encryption - where sensitive data is encrypted and routing information is left unencrypted.

Payload encryption can be implemented through asymmetric and symmetric techniques. Asymmetric encryption uses two keys - a public key for encryption and private key for decryption. This approach works best for a system with a small number of trusted subscribers who have access to the private key, and several possibly untrusted publishers. Asymmetric techniques, though more flexible, are generally more intensive with regard to computation, memory and bandwidth than symmetric techniques. These are always significant considerations for resource-constrained devices. Symmetric encryption uses the same key for encryption and decryption. It is simpler to implement and the measure of security is offloaded in a large part to the robustness of the key exchange mechanism (Katz, 2014).

Payload encryption ensures end-to-end data confidentiality in a constrained environment such as IoT, particularly in cases where TLS cannot be supported. In order to provide complete data transmission security, keys must be provisioned securely, with an authentication and an access control policy. These mechanisms can be offloaded to a central server that will transfer a significant amount of the processing away from the constrained devices.

2.2.3 Symmetric-Key Encryption

Symmetric-key encryption is an appropriate choice for use in an IoT-based payload encryption scheme, as long as sufficient key-exchange policies are enforced. There are three facets to consider in deciding on the symmetric-key technique to use - the algorithm itself, the mode of encryption and the key size. There are several different algorithms to choose from, for example AES, DES, 3DES, Blowfish, Camellia and ARCFOUR. These algorithms have different properties and optimisations making them suitable for various scenarios. Some of them, such as DES and ARCFOUR, have been rendered unusable due to technology improvements and detection of vulnerabilities within them (Kong et al., 2015). Each algorithm has permitted key sizes, and is implemented using one of the modes of operation outlined in Table 2.1.

The most common modes of encryption are detailed in Table 2.1. These encryption modes determine how each block is translated to ciphertext by the algorithm, and each varies in complexity and overhead (Almuhammadi & Al-Hejri, 2017). The Initialisation Vector (IV) is a string of bytes used to initialise some of the modes, and must be sent alongside the message to be used for decryption. Modes that use padding require message length to be a multiple of the block size, and padding is added to the end of messages to ensure this. These factors both lead to larger message sizes and additional bandwidth. A parallel architecture allows improved performance, and can generally be implemented by stream ciphers. Some modes of operation offer not only encryption, but also a mes-

Name	Cipher	IV (bytes)	Padding	Description	Parallelism	MAC
Electronic Code Book (ECB)	Block	None	Yes	Identical plaintext blocks encrypted into identical ciphertext.	Yes	No
Cipher Block Chaining (CBC)	Block	16	Yes	Plaintext block first XORed with previous ciphertext block.	No	No
Cipher Feedback (CFB)	Block	16	No	Plaintext block is XORed with output of encryption.	No	No
Output Feedback (OFB)	Block	16	No	Portion of encryption output is used as feedback for the shift-register.	No	No
Counter Mode (CTR)	Stream	8	No	Counter and nonce generate key stream, which is XORed with plaintext to produce ciphertext.	Yes	No
Counter with CBC-MAC (CCM)	Stream	Arbitrary length	No	Counter mode used for encryption with CBC-MAC for message integrity.	No	Yes
Galois/ Counter Mode (GCM)	Stream	Arbitrary length	No	High-speed counter mode encryption with GMAC for message integrity.	Yes	Yes

Table 2.1: Comparing symmetric key modes of encryption.

sage authentication code (MAC) for data integrity. This process is integrated within the encryption process, making it more efficient than providing the two functionalities separately.

2.3 Closely-related Projects

A significant amount of research is underway in the area of optimising security protocols over IoT networks and application protocols such as MQTT. With the present exponential growth and commercialisation of IoT, security has become a focal area of research. New protocols are being developed to suit the IoT framework, and standard techniques are being adapted. The crucial component of research regards the resource-constrained nature of IoT, and the aim to balance security with usability in such contexts (Nguyen et al., 2015).

Several areas of research focus on providing an adequate security framework for the MQTT protocol. MQTT presents its own challenges as a protocol, having a publish/subscribe architecture and few inherent security mechanisms (Yassein et al., 2017). As TLS is the recommended protocol for use over MQTT, some research looks into adapting this protocol for use in a resource-constrained environment. Other areas of research concentrate on using techniques such as payload encryption to provide lightweight end-to-end security. This can be implemented using a wide range of approaches, using symmetric, asymmetric or hybrid techniques.

A symmetric-based security framework is proposed by Shin et al. (2016), detailing an adaptation of the Augmented Password-Only Authentication and Key Exchange (AugPAKE) protocol for use over MQTT. The AugPAKE algorithm transmits authenticated session keys between the client and broker. Data is then transmitted with a secure symmetric-key encryption scheme using the established session key. The initial authentication of clients is simplified compared to TLS and the overall transmission overhead is reduced. However, this does not ensure end-to-end security as communication is only secured between client and broker.

A secure MQTT implementation is proposed by Singh et al. (2015), using asymmetric-key encryption. This uses lightweight attribute-based encryption (ABE) over elliptic curves. Using ABE, the data is encrypted based on an access control policy, and only approved clients can decrypt the data. This implementation does ensure end-to-end encryption, as only valid subscribers can decrypt the data, and though the broker has access privileges, it cannot decrypt it. An advantage of this approach is its support of broadcast encryption - where one encryption process is needed to deliver the message to all subscribers. The proposed protocol is secure against several common attacks. However, complex multiplicative inverse operations are necessary to implement ABE, incurring a significant processing overhead.

Rizzardi et al. (2016) propose a key management and policy enforcement framework for use in providing secure MQTT communications. The solution proposes an IoT middleware based on several Networked Smart Objects, using HTTP to communicate with the IoT devices. This enforcement framework requires devices to initially register with it in order for secure encryption to be provided. The devices are granted credentials, which can be used to authenticate themselves for future communications. The framework accepts key requests for devices to encrypt and decrypt data, which are granted based on authentication and the enforced policy. A similar system is proposed by Neisse et al. (2015), implementing SecKit, a security policy enforcement model. It offers a framework that can be applied to several IoT protocols, with an aim to decouple security from the everyday running of the system. This framework has been applied to the MQTT protocol (Neisse et al., 2014), and is executed as a broker extension. SecKit implements a context-based policy template that must be explicitly instantiated.

The solution proposed by Mektoubi et al. (2016) details a method to secure end-to-end MQTT communications with a certification authority (CA) generating certificates for both the client and topic. Asymmetric encryption techniques are used to implement this. The RSA and Elliptic Curve (ECDSA) algorithms are analysed for performance, noting that ECDSA uses smaller keys and RSA has a quicker execution time for data encryption. It goes on to offer a hybrid approach of the two algorithms, using RSA for data encryption and ECDSA for the digital signature. End-to-end encryption is provided, however, it does not adequately outline the transmission overhead involved, indicating that it may be comparable to the overhead incurred by TLS.

An asymmetric security protocol is proposed by Moustaine & Laurent (2012), focusing on the issue of authentication. This paper is based on RFID technology, which is extremely low-powered and concerned with operational overhead. It proposes authentication using an optimised implementation of NTRU, one of the fastest public-key cryptosystems. This implementation delegates the complex NTRU operations to a server, leaving the low-powered nodes to process only the lightweight operations such as additions and circular shifts. This system performs extremely well, with proven resistance to replay attacks and Man-in-the-Middle attacks with greatly reduced overhead. Its implementation over MQTT would mean that the broker must provide a dedicated server to process the NTRU calculations. This offloading of heavy computation away from the low-powered IoT devices should be at the forefront of design for an IoT system.

Several areas of research look into tailoring TLS to suit the IoT environment. It should be noted that due to the fact that TLS does not exist on the application layer, it is feasible for both TLS and payload encryption schemes to be used in tandem. Urien (2017) suggests Secure Access Modules for IoT. These are tamper-resistant microcontrollers to be added on either end of communication, implementing TLS/DTLS. These external pieces of hardware deal with the security features that are required, without draining the resources of the constrained devices. This allows a full TLS implementation to occur in an IoT network. However, an added piece of hardware would increase the cost of a device, which may not be feasible in many scenarios.

Another attempt at adapting TLS to an IoT context is proposed by Chung & Cho (2016). This approach uses fuzzy logic to adapt the TLS method in order to provide the most suitable and energy-efficient implementation. The system analyses the context, based on parameters such as the message length, residual device power and required security level in order to identify the most efficient ciphersuite to use. This solution primarily deals with the energy limitation, but does not address other constraints on the system, such as memory or bandwidth.

Katsikeas et al. (2017) analyse various AES encryption techniques over the MQTT protocol, in order to identify the most efficient regarding round-trip time and memory usage. Payload encryption was compared with link layer encryption, as well as comparing encryption modes with and without an accompanying MAC. Link layer encryption was found to have less of a performance impact due to the chip used, which has hardware accelerated AES encryption capabilities. However, it has a disadvantage in that it provides node-to-node encryption, rather than application-layer end-to-end encryption. AES-OCB, the authenticated encryption technique, was found to be the most resource-intensive, due to the complexity of the algorithm.

A payload encryption scheme is proposed by Jan et al. (2017) for use over the CoAP protocol. This is a lightweight mutual authentication scheme, by way of a four-way handshake mechanism. This is based on CoAP's client-server interaction model, aiming to improve upon DTLS, which is the most widely used security infrastructure in CoAP-based IoT communication. It aims to integrate security features into CoAP, rather than have them decoupled and existing as separate protocols. The authentication process aims to be as lightweight as possible, only requiring a small number of request/response interactions to exchange a session key and begin communication. Pre-shared secrets are used to verify the identity of legitimate system entities. It shows significant improvements over DTLS in handshake duration, memory consumption and average response time.

Due to the growing requirements of IoT, a robust, scalable and lightweight security mechanism is required for MQTT, the most prevalent IoT application-layer protocol. Though TLS may be the most straightforward security implementation, there are several reasons why it is not suitable for resource-constrained devices that comprise IoT. There is a significant amount of research going into other possibilities. These solutions aim to reduce transmission overhead on the security layer, reduce the necessary computation and memory requirements for TLS implementation, or deal with various insecurities related to TLS over MQTT.

2.4 Problem Formulation

Upon analysis of the background and research involved in this area, it is apparent that security is a significant concern in IoT. The monumental growth and commercialisation of IoT has lead to the deployment of several insecure systems. This is due to the fact that implementing security is resource-intensive, and can be a difficult and expensive endeavour to apply to low-powered devices. It is therefore not always seen as a worthwhile pursuit. A study by HP Fortify (2014) found that 70% of IoT devices contain vulnerabilities, including password security, encryption and general lack of granular user access permissions. These insecure systems pose a threat, not just to their users, but to the industry in general. IoT suffers from a loss of trust, and the perception of vulnerability due to its security shortcomings (Sicari et al., 2015). Therefore, the development of an appropriate solution to this problem is a necessary and worthwhile undertaking.

The problem to be addressed is how to apply a security scheme efficiently in an extremely resource-constrained environment, using a publish/subscribe communication framework. This encompasses a many-pronged problem. First of all, the design must accommodate for low-powered devices with limited capabilities. Processing on the device-end should be reserved for essential functionality, and this should be optimised for minimal overhead. An IoT device aims to remain in idle state as much as possible to preserve resources, and aims to transmit messages with minimal bandwidth. Secondly, the design should take the MQTT architecture into consideration, providing end-to-end encryption from publisher to subscriber. It should improve upon TLS with regard to resource consumption and providing within-broker security. Finally, the scheme should be robust in its security mechanisms, ensuring that confidentiality, integrity and availability are maintained throughout.

Chapter 3

Design

This chapter gives an outline of how the problem formulation described in Section 2.4 has been translated into a solution. Section 3.1 gives an initial overview of the design, outlining the high-level aims and purpose of the solution. The logical architecture of the scheme is illustrated in Section 3.2, describing the system components, their role and how they fit together. In Section 3.3, a sequence diagram is used to demonstrate the inner processes involved when the scheme is in use. The most significant processes are expounded within Sections 3.4, 3.5 and 3.6 - detailing the encryption, authentication and access control and key caching designs respectively. These functions are fundamental to the successful and efficient implementation of the scheme within an IoT system. Section 3.7 details the data architecture of the scheme, displaying the packet structure used and defining the contents of the header and payload. Finally, the design is summarised in Section 3.8.

3.1 Overview

A secure and efficient transmission architecture has been designed to serve a resource-constrained environment over the MQTT protocol. This design aims to minimise the processing, bandwidth and memory overhead involved in encryption on resource-constrained devices, as well as minimising the number of necessary MQTT messages to be sent and received by the device. The design provides confidentiality and integrity to IoT messages using symmetric-key payload encryption. Authentication, access control and key management are provided through a Key Management Service (KMS). MQTT's publish/-subscribe architecture necessitates asynchronous communication between clients through the MQTT broker. The design ensures end-to-end encryption, so that sensitive information stored by the broker is encrypted and message data cannot be compromised through a broker attack.

Several symmetric-key encryption algorithms were compared to identify the most suitable encryption for the IoT environment, with regard to processing power and message overhead. It was determined that AES (Rijndael) should be used in counter mode with CBC-MAC (CCM), using a 128-bit key. The payload of each message sent between clients in the system is encrypted and decrypted using this algorithm. The MAC is then used to assure message integrity, identifying if the packet has been tampered with in any way. Header information is not encrypted, so that the packets can be routed accordingly without the need to decrypt them at each node.

The KMS provides keys to publishers and subscribers after authentication, based on a defined access control policy. All clients in the system must be initially registered in a secure manner with the central KMS, wherein they are provided with a private key and private topic that only they (and the KMS) have access to. They are also assigned a client ID and password. When a client requests a key, either to encrypt or decrypt, they are first authenticated by the KMS using these credentials. The requested key is then encrypted by the KMS using the client's private key, and sent to the private topic for the client to receive. The client uses their private key to decrypt the requested key, which can then be used to encrypt or decrypt their message. Each encryption key expires after a certain amount of time, and the publisher must request a new key when this occurs.

The various design choices were made with the intention of providing a lightweight security framework for use in resource-constrained IoT contexts, and improve upon the use of TLS with MQTT, which is often unsuitable in these scenarios. The balance between security and usability was at the forefront of the design. Optimisations have been made to offload any resource-intensive processing away from the device and towards external capabilities. The purpose of this system is to ensure secure and efficient transmission of IoT messages over the MQTT application-layer protocol. It is noted that security concerns exist on all layers, and message transmission security does not necessarily ensure full-stack network security.

3.2 Logical Architecture

The logical architecture seen in Fig. 3.1 represents a high-level design of the system, identifying the essential components involved and the dependencies between them. It depicts a classic MQTT architecture with publishing and subscribing entities communicating asynchronously through the central broker. In addition to this is the KMS, which is essentially a publisher and subscriber providing a management service to the devices involved in the system. As MQTT has a centralised architecture, each component has a

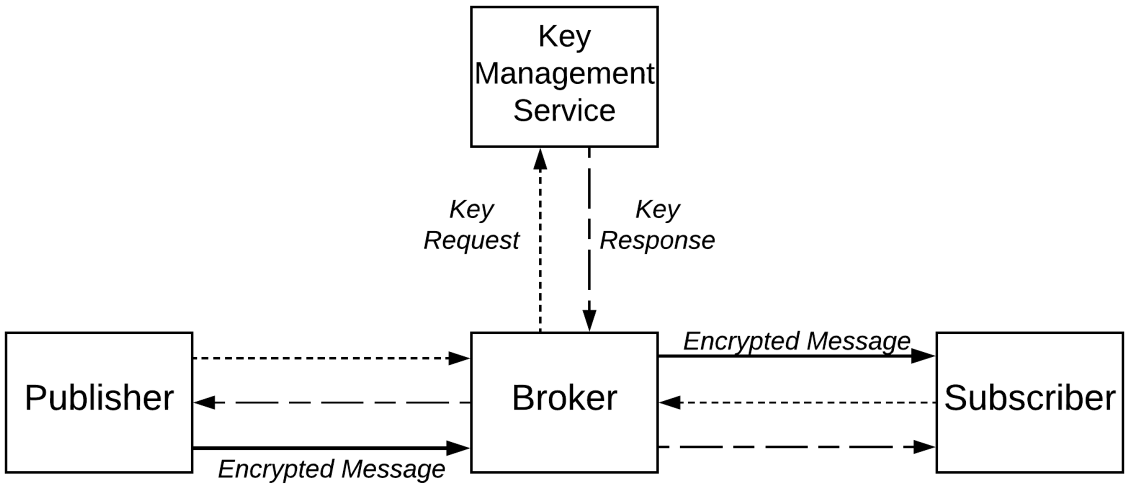

Figure 3.1: Logical Architecture.

direct dependency only with the broker.

3.2.1 Publisher

The publisher sends messages to topics on the broker to be forwarded on to the subscriber. When a message is to be sent to a topic, it first checks the validity of its encryption key for that topic in its key cache. If it does not exist, has expired, or is invalid due to length or corruption, a request is made to the KMS to obtain a new key. The response is first decrypted using the private key, then the key is used to encrypt the payload of the message. The corresponding key ID is sent within the packet header in plaintext. If the payload exceeds the MTU of the underlying protocol, the message will be fragmented and sent in multiple packets. Depending on the Quality of Service (QoS) level of the MQTT configuration, the publisher may receive acknowledgement from the broker upon receipt of the packet.

3.2.2 Subscriber

The subscriber is an entity that subscribes to topics on the broker and is forwarded any message that is received on these topics. When a packet has been received, it is parsed to identify the various packet elements. If large messages have been fragmented and arrive in multiple packets, the subscriber waits for all fragments before reassembling the full message. The decryption key ID is extracted from the header and the subscriber checks its key cache to see if it contains the corresponding key. If not, it must request this key from the KMS. The decryption key response is received and is decrypted with the private key. This received key is then used to decrypt the payload of the original message, and message integrity is established using the MAC. The subscriber can then process the plaintext message accordingly.

3.2.3 Broker

The broker is an impartial component that performs routing based on the MQTT publish/subscribe protocol. The broker can host any number of topics for the publishers and subscribers to communicate through. It can implement an access control policy on these topics, using a username/password whitelist. It also provides some defined topics for communication between the devices and the KMS. These include topics for both requesting an encryption key and requesting a decryption key. It also provides private topics, establishing a dedicated asynchronous channel between the KMS and each particular device for key exchange.

3.2.4 Key Management Service

The KMS provides key management, authentication, monitoring and access control for the system. As symmetric-key encryption is being used, the same key is required to encrypt and decrypt the data. This service provides such functionality in a secure and easy-to-manage way, while also providing other essential capabilities so that they do not need to be performed on the device itself. Communicating over MQTT and existing in parallel with the broker, it fits seamlessly with the MQTT paradigm and conforms to the asynchronous nature of the system. The KMS monitors and analyses all traffic passing through the system.

Authentication is established by means of a client ID and password for each involved entity (publisher, subscriber or both), which must be checked and confirmed for every request. Upon initial registration of the entity, the access policy is determined on a more granular level. The KMS can restrict a device based on publishing and subscribing rights, as well as on the particular topics they can access. Each request is reviewed based on this access policy, and keys will only be provided to entities that are approved for their requests.

The KMS subscribes to two key request topics - one for encryption keys and one for decryption keys. Publishers request encryption keys to encrypt messages they wish to send. Subscribers request decryption keys to retrieve the original key that was used to encrypt the message they have received. These are all stored securely by the KMS once they have been issued. When an entity has been authenticated, its private key is retrieved and is used to encrypt the key to be sent. This encrypted message is then sent to the device through its private topic.

3.3 Sequence of Events

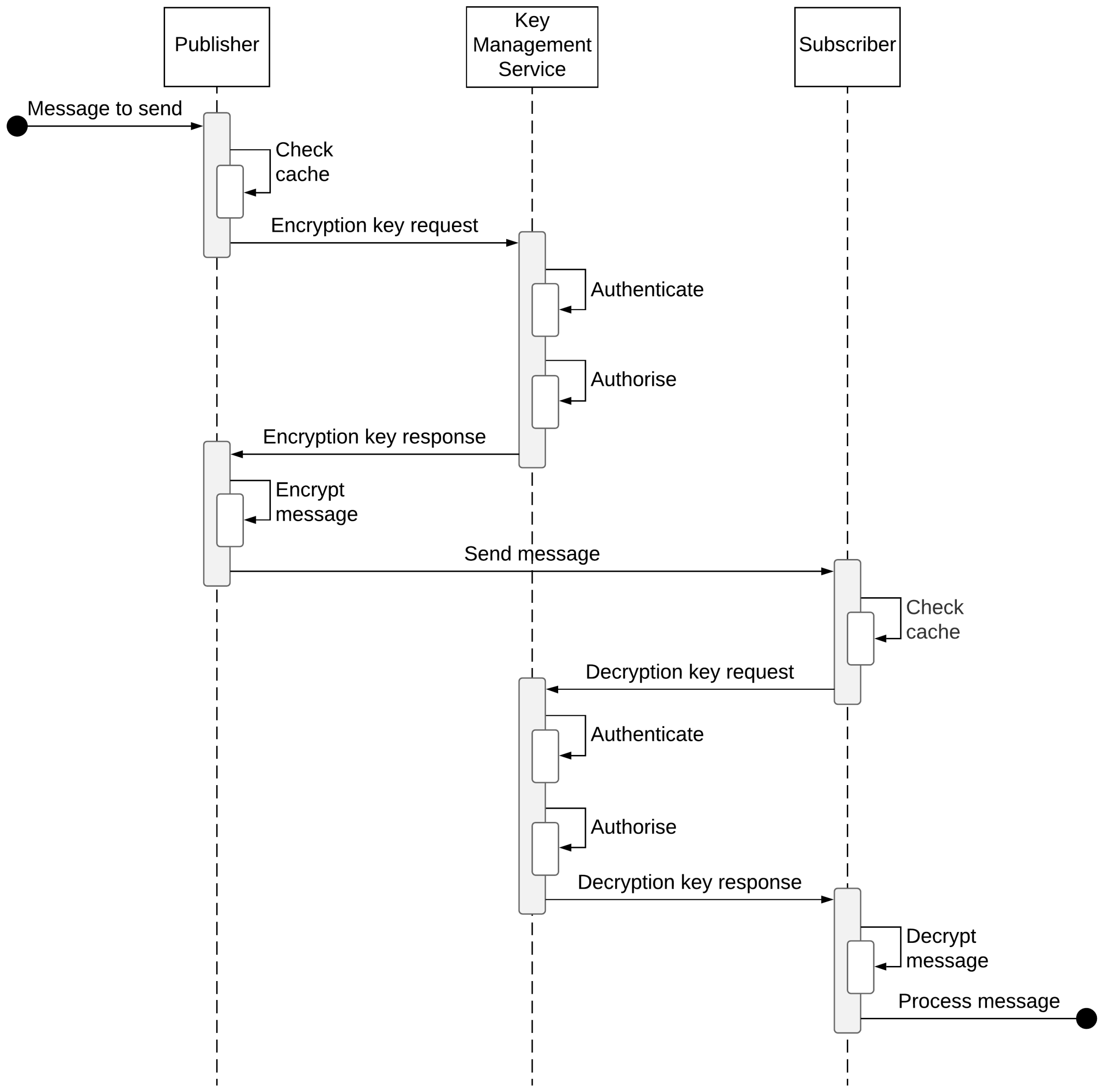

Figure 3.2: This sequence diagram outlines the process involved when a publisher wishes to send a message to a topic. This scenario assumes neither publisher nor subscriber hold the requisite keys in their respective caches. The broker has been omitted from the diagram for clarity.

- Publisher wishes to send a message to a topic.

- Publisher checks if the cached key for the topic is still valid. If so it can be used to encrypt the message.
- If the key has expired, an encryption key request is sent to the corresponding topic.
- This request is forwarded to the KMS by the broker and decrypted using the publisher's private key.
- KMS authenticates the publisher.
- KMS checks that the publisher has permission to publish to the requested topic.
- KMS creates a new encryption key for the publisher and stores the relevant information.
- KMS uses the publisher's private key to encrypt the new key data, and sends it through their private topic.
- Publisher receives this response and decrypts it to obtain the encryption key.
- Publisher uses this encryption key to encrypt the message.
- The packet is constructed with all relevant header information, and one or more fragments are sent to the topic.
- Subscriber receives these packets, reassembles them and extracts the header information and encrypted message.
- Subscriber checks its key cache using the key ID sent in the message header. If present, it is used to decrypt the message.
- If not, subscriber sends a decryption key request to the KMS, using the key ID.
- KMS receives the request, authenticates then verifies the subscriber against the access control policy.
- The key ID is used to retrieve the required key, which is then encrypted with the subscriber's private key.
- KMS sends the encrypted message through the subscriber's private topic to the subscriber.
- Subscriber decrypts the response, and uses the obtained key to decrypt the original message from the publisher.

3.4 Encryption

Confidentiality is ensured by the system using symmetric-key payload encryption, specifically AES CCM mode encryption with a 128-bit key. AES is the Advanced Encryption Standard, an algorithm established by the National Institute of Standards and Technology (NIST) in 2001. It uses the Rijndael cipher, allowing key sizes of 128, 192 and 256 bits. This is by far the most widely used symmetric-key algorithm, and there is no known attack that would allow someone without knowledge of the key to access AES-encrypted data (McKay et al., 2017). Due to its broad use, it is supported by a wide range of security packages. For these reasons, this was the algorithm chosen for use in the design.

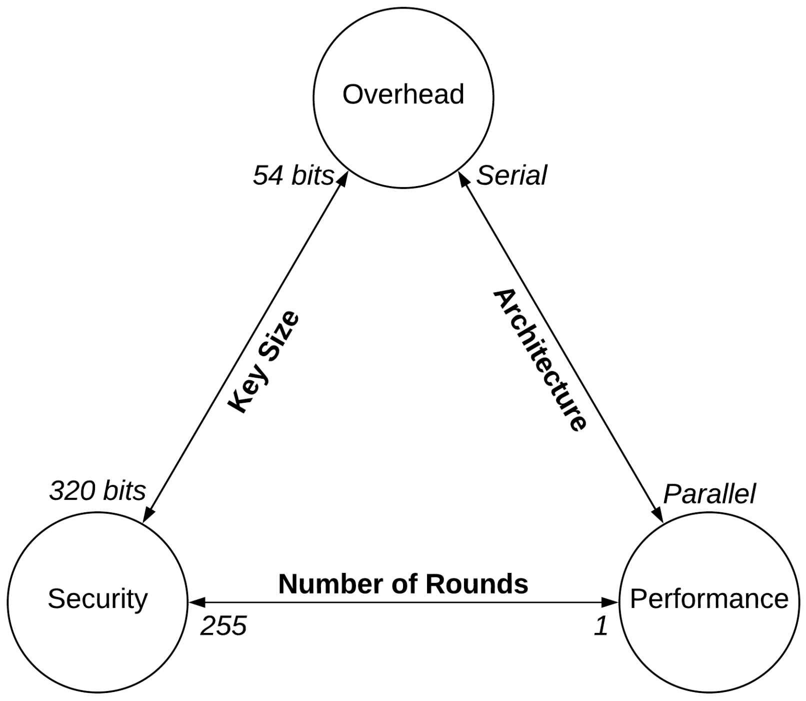

Figure 3.3: Relationships between the three security qualities to consider for symmetric key encryption.

The implementation of the algorithm should be optimised by considering the three vertices of the security relationship illustrated in Fig. 3.3. The AES key sizes of 128, 192 and 256 bits correspond to 10, 12 and 14 rounds respectively. As seen in Fig. 3.3, a smaller key size and number of rounds relates to superior performance and overhead, yet detracts from the level of security provided. These are all extremely important considerations in the system design. The practical limit of breakability for symmetric key sizes is 80 bits with current technology (Kong et al., 2015). As all AES keys are above this limit, they are all unbreakable through a brute-force attack. Similarly, the extra rounds do not linearly produce a more secure system, yet they make the encryption process 40% slower (Katz, 2014). For these reasons, it was concluded that AES-128 would be sufficient.

CCM mode proved to be the most suitable mode of encryption to use, for several reasons. It is intended for use in a packet environment, which is ideal for IoT. This mode combines the counter encryption mode with a CBC-MAC for message integrity. Counter mode was found to be the most efficient and suitable encryption-only mode. This mode transforms a block cipher into a stream cipher by encrypting each block using an incrementing counter. Due to the fact that counter mode is not a block cipher, is does not require padding to fill remaining bytes in the block. This reduces the number of bytes to be sent, reducing the average packet overhead. A random 8-byte nonce (unique Initialisation Vector (IV)) is used to initialise the counter to a unique value. This combination must only be used once with the same key, and is incremented for each block that is encrypted. The nonce is sent in plaintext along with the encrypted message in the packet. This improves upon other modes of encryption that use a 16-byte IV, as it is only half the size.

The CBC-MAC uses cipher-block chaining to produce a MAC tag, which can be used to ensure the message is authentic. This MAC adds 8 bytes to the message length, and CCM mode calculates it on the plaintext before encrypting the whole message, including the MAC. This is in comparison to using a separate CBC-MAC or HMAC function to provide integrity, which would entail passing around another key and is less efficient than CCM. Galois/Counter Mode (GCM) is another encryption/MAC hybrid technique, though it is less commonly found and it is much more resource-intensive. Its expensive operation can be accelerated using previously computed lookup tables, though this still requires a significant amount of memory (Szalachowski et al., 2010).

An MTU is often set by an underlying protocol, limiting the maximum size packet that can be sent by a device. When a message is being prepared to be sent, the entire length of the message (headers and payload) is identified, and compared with the MTU. If the length is larger than the MTU, the packet will fail to send and must be fragmented. In the case of fragmented messages, only one security header is necessary for the entire message. A fragmentation offset is stored in the first byte of the security header. The subscriber will store all fragments that are received, until the fragmentation offset indicates that it contains the final fragment. The first fragment contains the security header, and the payloads of all other fragments are concatenated and decrypted accordingly.

3.5 Authentication & Access Control

Authentication and access control are policies enforced primarily by the KMS, with added capabilities enforced by the broker. The perspective of the system is of a distributed IoT system. This involves a known number of clients that are authorised to access the data, and actuate some function on the system. Clients can be both publishers and subscribers within the MQTT paradigm. There are also a number of topics relevant to the system, with restrictions on entities that can publish or subscribe to these topics. The policies are stored and configurable within the KMS, and encryption and decryption keys are only entrusted to those entities that have been approved. Each key is associated with a particular topic, and the KMS ensures that only authenticated and authorised clients for that topic can access the key. It also ensures, through system monitoring, that the key is only being used for approved operations.

Upon initial registration of an entity within the system, its individual access policy is established. The client ID and hashed password are stored within the KMS as a trusted party, and these are used to authenticate the client upon the receipt of a request before any further processing is carried out. Along with this identification, the KMS stores whether the entity has publishing rights, subscribing rights or both. Any topic used by the system is stored by the KMS, and a granular restriction level can be set on particular entities accessing particular topics. A hierarchy of access limitations can be enforced by configuring the KMS to the needs of the system. Another layer of security can be enforced by the broker itself, in that PUBLISH and SUBSCRIBE requests can be denied by it if the entity is not on an approved whitelist for such a request.

3.6 Key Cache

An in-memory key cache is used for clients to store recently and regularly used keys for quicker access, instead of issuing key requests for every message to be sent and received. Two types of key cache are used, one for encryption keys and one for decryption keys. These are implemented in slightly different ways, and can vary in caching policy. Both key caches are created using synchronised hash tables. The encryption key cache stores keys based on topic, so that each publisher uses one key per topic until it expires or is otherwise invalid, after which a new key is requested for the topic when it is needed. This uses a time aware least recently used (TLRU) cache replacement policy, in order to remove keys for topics that are not used very often, being aware of their expiry time.

The decryption key cache stores keys based on the key ID. It uses a least recently used (LRU) cache replacement policy. This differs as decryption keys do not technically expire, as old messages encrypted by an expired encryption key must still be able to be decrypted. However, these expired keys cease to be regularly used, and can be removed from the key cache when newer keys are obtained. It is likely that the decryption key cache will be significantly larger than the encryption key cache. This is because it is likely for a client to publish to a small amount of topics, and to receive messages from several different topics and several different clients. The system can define the maximum size of the caches, depending on resources available to each client.

3.7 Data Architecture

The vast majority of packets sent between the components in the system are MQTT PUBLISH messages. These packets follow a defined structure, with a fixed header, variable header and payload (Fig. 3.4). An additional security header is implemented at the start of the payload in the design. This allows messages to communicate necessary security information in plaintext, while having the sensitive data encrypted in the payload. One of the fundamental aims of the system is to minimise the packet overhead and bandwidth, in that a minimal number of packets should be sent, and each packet should contain as few bytes as possible. For this reason the data architecture is optimised to its fullest as part of the design.

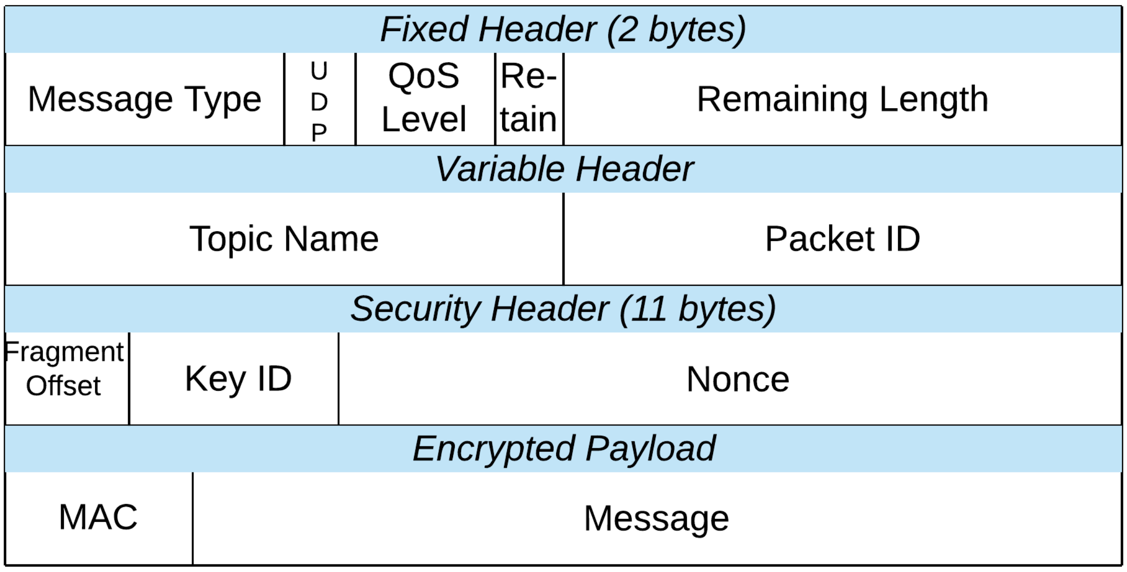

Figure 3.4: Message format of MQTT PUBLISH packet used within the scheme.

The fixed header is two bytes long. The first byte defines the packet type (PUBLISH in this case) and flags identifying the QoS level, duplication and retain instructions. The second byte records the remaining length of the packet. The variable header contains the topic name and the packet identifier (only necessary when QoS is greater than zero). The security header is 11 bytes long. The first byte represents the fragmentation offset.

The following two bytes contain the key ID of the key used to encrypt the payload. The final 8 bytes are used to communicate the nonce, which must be used in the encryption and decryption process. Finally the encrypted message, along with the encrypted 8-byte MAC, is sent in the remaining bytes in the payload. In case of fragmentation, the first packet contains the full security header, and all following fragmented packets will only hold their fragmentation offset in the security header.

3.8 Summary

A high-level overview of the scheme design has been outlined in the above chapter, which will be expanded upon and detailed within Chapter 4, Implementation. The design specifies how the system components communicate with one another, and the processes involved in the typical operation of the system. It indicates the purpose of each component and why they have been implemented in such a way. The core security functionalities are outlined, detailing how the CIA triad is realised to produce a secure system. Confidentiality and integrity are provided with the encryption and MAC techniques described in Section 3.4. These are operations that occur on the device itself. The majority of the availability functions are provided by the KMS, which equips the system with authentication, access control and key management. Availability is aided by the key cache (Section 3.6), allowing immediate encryption for regularly used keys and approved communication. The design encompasses a secure, efficient communication mechanism for IoT systems.

Chapter 4

Implementation

This chapter outlines the actual work that has been carried out in implementing the design laid out in Chapter 3. Section 4.1 presents a high level view of the implementation, defining the architecture of how all system components have been deployed, and how they interact with one another. Section 4.2 details the broker configuration used in order for all components to communicate through it. Section 4.3 describes the implementation of the Key Management Service, deployed as a resource within the MQTT framework. Section 4.4 defines the C library that has been developed to provide capabilities to the resource-constrained IoT devices. These capabilities have also been provided for resource-unconstrained system components in the form of a Python module, outlined in Section 4.5. Finally, Section 4.6 summarises the end-to-end implementation and deployment of the scheme.

4.1 Deployment Architecture

The scheme outlined in Chapter 3 has been deployed as an end-to-end IoT package, encompassing all necessary components to be easily integrated into an IoT-over-MQTT application (Fig. 4.1). This scheme has been abstracted to library form, to provide secure transmission functionalities to the various application components. The necessary MQTT configuration has been outlined and identified, using Eclipse Mosquitto as message broker. The Key Management Service (KMS) is deployed as an autonomous utility providing services to the system components. A C library has been developed to provide the capabilities required for the resource-constrained IoT devices. Finally, utilities have been provided, in the form of a Python module, for resource-unconstrained components participating in the IoT system.

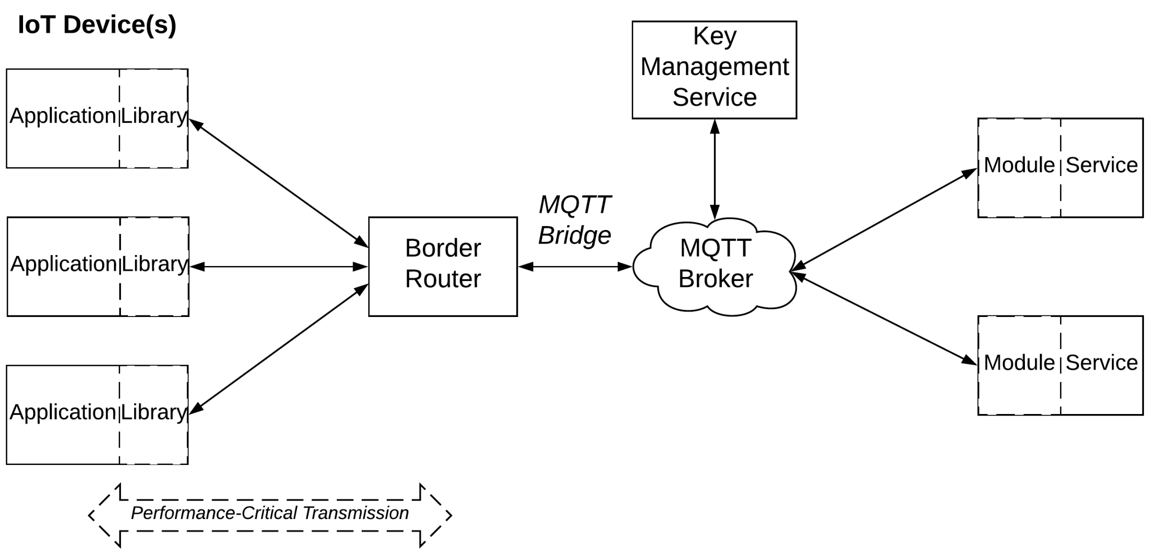

Figure 4.1: Deployment Architecture Diagram.

4.2 MQTT Broker

Mosquitto is a lightweight, open source implementation of an MQTT broker that is suitable for use on all types of devices[1]. It requires very little configuration, merely listening on port 1883 for any MQTT packets to be dealt with accordingly. Topics do not have to be previously configured on the broker to be published or subscribed to. Rather, if the broker encounters a request for a previously unencountered topic, it will provision resources for it and route any subsequent messages for the topic appropriately.

MQTT Bridge

The broker deployment, as seen in Fig. 4.1, is implemented using a border router in the IoT device environment, forwarding MQTT packets to and from the central broker hosted on the cloud (AWS). Bluetooth Low Energy is the network protocol used in this IoT implementation, which has a range of approximately 100m. The border router, hosted on a local Linux virtual machine, runs an identical Mosquitto broker that picks up the MQTT packets over Bluetooth. An MQTT bridge is configured between the two brokers. This essentially allows one of the brokers to act as a client to the other, publishing and subscribing to all relevant topics. Some simple configuration is done on one of the brokers' `mosquitto.conf` files, identifying the address and port of the remote broker, along with any topic specifications or remapping necessary. This essentially implements

[1]https://mosquitto.org/

MQTT packet forwarding.

Security

In terms of security, the broker is capable of implementing an authorisation policy. In Mosquitto, settings can be changed in the `mosquitto.conf` file to enforce limitations on client access. This is a desirable feature for most IoT applications, to ensure that only authorised clients can connect to the system, and be able to publish or subscribe only to their relevant topics. A password file can be stored in the broker's `conf` file that stores approved clients. The MQTT CONNECT packet contains optional `username` and `password` fields, which are checked against the password file before connection is approved. The broker can also store an access control list (ACL) file, similarly implementing restrictions on a more granular level, specifying approved topics, operations (publish, subscribe or both) and QoS levels. This is based on the client ID that can be sent in the header of MQTT packets.

These measures add a weak layer of security to the system in place. However, a secure transmission channel does not exist between client and broker in this implementation, and the MQTT header is transmitted in plaintext. For these reasons, packets could be intercepted in transmission and the relevant authorisation criteria could be gleaned and spoofed to gain access to the broker. The KMS provides a much more secure means of ensuring authorisation and access control.

4.3 Key Management Service

The KMS is deployed as a central server communicating entirely over MQTT, directly with the cloud-based MQTT broker. It is responsible for ensuring secure key exchange, key life-cycle management, client authorisation and access control. Though the communication protocol used is MQTT instead of HTTP, the service has been designed to emulate a RESTful API in several ways. The KMS provides resource endpoints through the use of topics. The entire scheme is constructed on an event-based architecture, where minimal processing is done until an event triggers the appropriate response. This is reflected in the stateless nature of the KMS, where each request is dealt with accordingly, before returning to idle state. This design is adapted to fit the asynchronous, decoupled request-response nature of MQTT systems.

Registration

A design decision that has been enforced on the scheme is that any client participating in any way must first be registered in a secure manner with the KMS. This process should be done offline by the system administrator, in a way that does not transmit any of the sensitive information over the network. The client is assigned a client ID and password that it can use to identify itself to the KMS (with the KMS storing the SHA-1 hashed password). It is also assigned a private topic for direct communication from the KMS, and a private key used to decrypt any responses received from the KMS. Both entities store these four pieces of data, which are essential for ensuring secure communications throughout the system. It is assumed for this implementation that all devices in play have been successfully registered.

Requests

The basic request endpoints provided by the KMS are encryption and decryption key requests. An encryption key is required by a publisher to publish a message, and a decryption key is required by a subscriber to decrypt a message it has received. Requests are filtered based on the topic they have arrived through. The client is first authorised and approved for access regarding the particular request. In the case of an approved encryption key request, a new key is created by the KMS. This is sent back to the publisher through their private topic, along with the key's ID and expiration time. These values are stored by the KMS, along with the client ID of the requester. An approved decryption request will include the ID of the key it wishes to obtain. If this is successfully retrieved by the KMS, it is sent back to the subscriber through their private topic.

Encryption and Decryption

Each request is encrypted by the client using its private key for transmission. The client ID is left unencrypted in the header of a request, which is used by the KMS to locate that client's private key. This is then used, along with the nonce sent in the header, to decrypt the request. Similarly, the response is encrypted with the same key and a new nonce, to be returned to the client through their private topic. The same encryption technique, AES-128 with CCM Mode, is used throughout the scheme for encryption and decryption. Since MQTT payloads are always binary, its not necessary to encode the encrypted message, a raw byte array, to a textual representation such as base64. This saves on additional bandwidth, as base64 encoding is more bloated, using 4/3 times as many bytes.

Authorisation and Access Control

The KMS must verify that each request is valid with regard to the application's access policy. Authorisation is carried out by means of the client ID and password established upon registration. The client ID is stored in the request header, and the password is stored within the payload of the request that is encrypted. This implements a form of two-factor authentication, in that the requester must have both the private key in order to properly encrypt the packet, and the password itself. The password is not sent directly, as this should only be known by the client. A cryptographic hash, SHA-1, is instead used, and this is the value that is stored in the KMS to compare and authenticate the client.

Once the client has been authenticated, the rest of the request is analysed. Each request must include the intended topic, and QoS level in the case of publishing. An access control list, similar to the broker's ACL, is stored by the KMS. Upon receipt of an encryption key request, the KMS checks whether the client has publishing rights for the specified topic and QoS level. If access is granted, the key issued is associated with those particular criteria. When a decryption key request is received, the client is first checked to make sure that they have subscribing rights to the topic in question. It is then ensured that the key requested has been associated with the correct topic. This is to prevent topic spoofing, where the publisher or subscriber requests a key for a topic that is not the intended one, that perhaps they do not have access rights to.

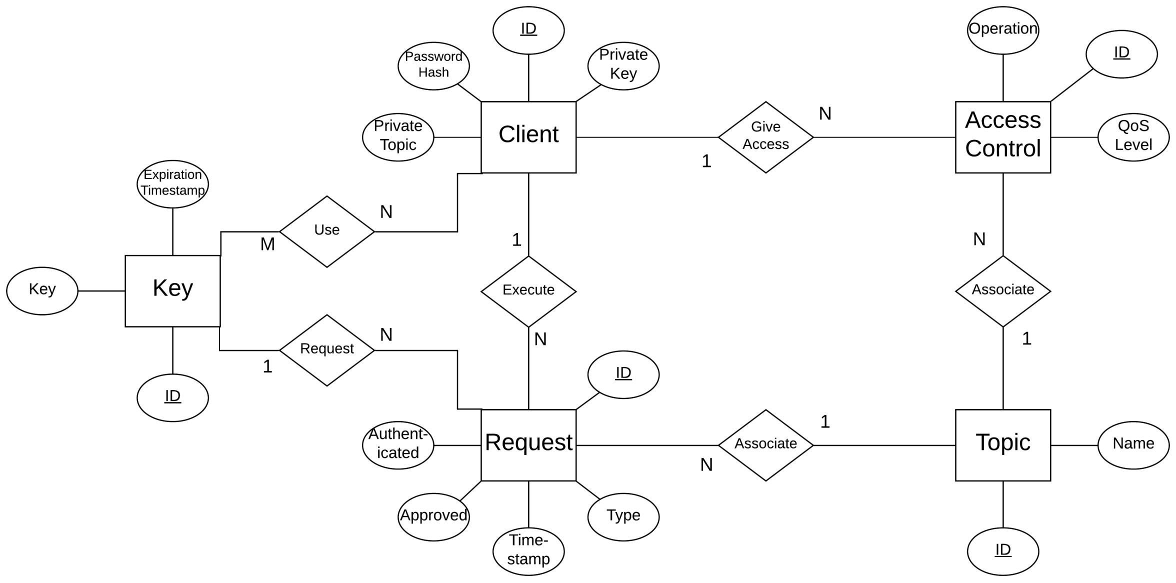

Figure 4.2: Entity Relationship Diagram for the KMS.

In order for these policies to be enforced, a relational MySQL database was implemented to store the relevant persistent information, as seen in Fig. 4.2. Whenever a client is registered onto the system, a row in the `Client` table is added. The `Access Control` table stores all information on the rights and restrictions of each client. Any topics in use are stored in the `Topic` table. Both the `Access Control` and `Topic` tables should be synchronised with the broker's ACL and topic list to ensure there is no mismatch. All requests made to the KMS are stored, along with whether they have been authorised and approved. Tracking of requests can help to identify potential security breaches and trigger an alert in these cases. When an encryption key request leads to a new key being created, it is stored in the `Key` table. Finally, a table is implemented that stores all clients that have accessed each particular key, for encryption or decryption.

Monitoring

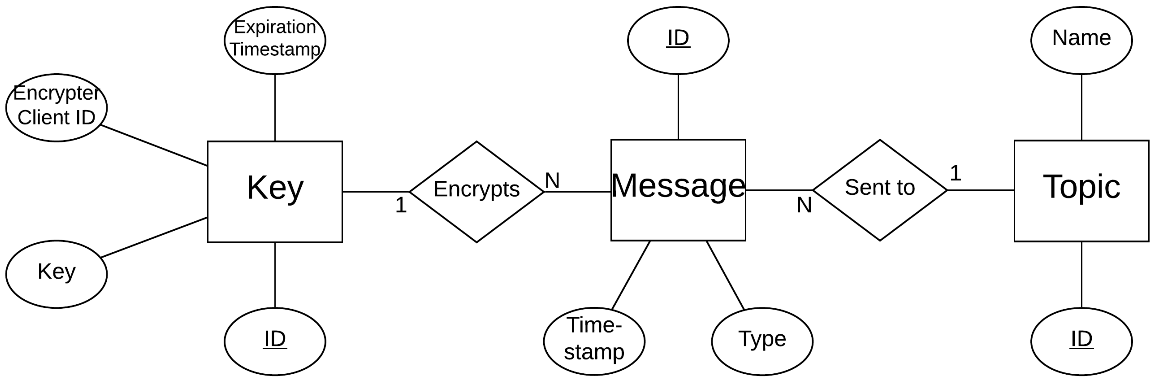

Figure 4.3: Entity Relationship Diagram for KMS Monitoring.

Along with the data represented in Fig. 4.2, the KMS also stores information on every message passing through the system (Fig. 4.3). In order to do this, the KMS subscribes to all topics supported by the broker. The actual payload of the message is not decrypted or stored, but information such as the topic it was sent to, time it was sent and key ID are stored. The key ID can be used to link the message to the client who sent it. These messages are passively monitored in order for the KMS to be aware of the operation of the system. Monitoring can identify any unusual system operation, such as a client suddenly stopping its transmission or an influx of unexpected messages. In this way security attacks such as DoS attacks can be identified, thwarted and alerted to as quickly as possible.

4.4 IoT Device Library

The IoT device implementation has been deployed as a C library, providing secure communication capabilities to a sample application running on a microcontroller board. Functionalities of the system include MQTT communication, payload encryption of messages and secure key exchange. The decision was made to develop such a system on a microcontroller rather than a simulator (Cooja, Omnet++) or emulator (QEMU, Mininet) for several reasons. Simulations are the most straightforward way to implement a system such as this, however, they are likely to be unrealistic and not account for real-world complications. Emulators will produce a more practical and realistic operation of an IoT system. Due to the varied and often unpredictable nature and environment of IoT, it was anticipated that these implementations would not adequately identify issues that may arise. The aim of producing such a system was to determine the most suitable security protocol for a constrained environment, and in order to do so the implementation should reflect a real-life IoT system to the fullest extent possible.

Technologies

IoT device development has been carried out on a Nordic Semiconductor NRF52 Development Kit[2]. This microcontroller integrates a low power ARM Cortex-M series microprocessor with memory and peripherals into a single microcontroller package, including a 2.4GHz radio designed for Bluetooth Low Energy communication. The Nordic Semiconductor nRF5 SDK is used on the microcontroller as the BLE "softdevice" for development. SEGGER JLink software[3] has been used for flashing and debugging the device, along with command line tools provided for the Nordic Semiconductor nRF5x range. The GNU ARM Embedded Toolchain[4] provides compiler and linker functionalities to the development system.

The operating system employed by the device is Zephyr[5], a real-time operating system (RTOS) designed for use on resource-constrained devices. Zephyr is an open source, highly configurable RTOS that is optimized for small memory footprint devices with security in mind. Functionality is provided for MQTT communication, with support for BLE and 6LoWPAN. On top of this, a security library, mbed TLS[6], is used to provide cryptographic functionalities such as encryption and hashing. These systems are written in the C programming language, and therefore the library and application developed is also written in C. Finally, in order to debug and display system outputs from the board, a serial port terminal application, CoolTerm[7], is used.

[2]http://infocenter.nordicsemi.com/index.jsp
[3]https://www.segger.com/downloads/jlink
[4]https://developer.arm.com/open-source/gnu-toolchain/gnu-rm
[5]https://www.zephyrproject.org/
[6]https://tls.mbed.org/
[7]http://freeware.the-meiers.org/

Application

An application was deployed to run on the microcontroller board, providing a means of testing and analysing the functionalities developed. This application provides a regular stream of messages to be encrypted and sent to the broker. The content, length, number and delay between messages can all be varied, in order to test the robustness and performance of the system. It also subscribes to a particular topic, which is used as an input endpoint. A user can publish any message to this topic, which the application will receive, encrypt and send over MQTT.

```
static void handle_input_msg(char* msg) {
        if (starts_with(‘‘rand’’, msg)) {
                msg = msg + 5;
                int rand_length = atoi(msg);
                char* rand_str;
                rand_string(rand_str, rand_length);
                encrypt_and_send(rand_str);
        } else {
        encrypt_and_send(msg);
    }
}
```

Listing 4.1: Handler function for messages received on the input topic. User can input a message of any length to be encrypted and sent. The user can also input the keyword `rand` followed by a number to send a random string of the specified length.

Encryption and Key Request Process

Fig. 4.4 depicts the steps undertaken when an application requires a new message to be published. Functionality is provided for the device to be both a publisher and subscriber to topics. For the purpose of description, the encryption and publishing process will be outlined for the IoT device, and message receipt from subscription and decryption process will be outlined for the resource-unconstrained service (section 4.5). This is based on the conventional perspective of IoT devices such as sensors publishing information about their environment, which is then utilised by services to make sense of the data provided. It is also noted that systems operate bi-directionally, and services can publish messages in order to actuate some function on the IoT device.

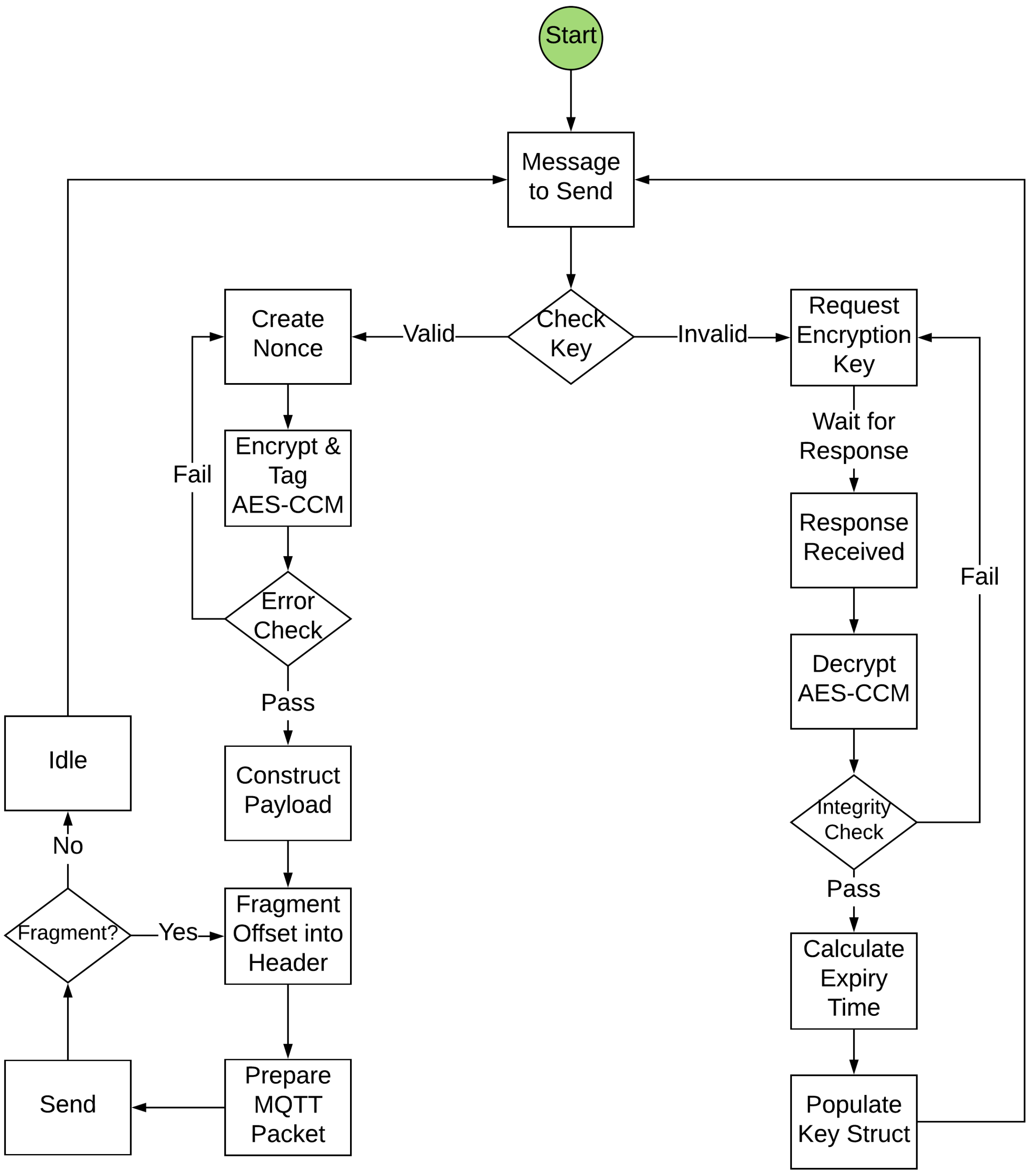

Figure 4.4: Encryption and Key Request Process

Message to Send. The aim of most IoT systems is for the application to lie in idle state until some event (a timer, change in environment, received message etc) triggers a response. The response usually requires some form of communication. When this arises, the application will call a function from the library to send the message.

Check Key. The first step in the process is to verify whether the encryption key currently in use for the topic is valid. The first check is on whether the key exists. When a device starts initially or restarts, it will not have any keys stored in the key cache. If one does exist, the expiration time is checked to make sure it has not expired yet. Finally, it is checked to make sure the key itself is valid, in that it is the correct length and format for the encryption algorithm.

```
static bool check_key(char* topic) {
    key_struct* encryption_key;
    int rc = hashtable_get(key_cache, topic, &encryption_key);

    if (rc != 0) {
        // Key has not been received yet
        if (!requesting)
            request_encryption_key(topic);
        return false;
    }
    if (k_uptime_get() > encryption_key->timestamp) {
        // Key has expired
        request_encryption_key(topic);
        return false;
    }
    if (strlen(encryption_key->key) != KEY_LENGTH) {
        // Incorrect key length
        request_encryption_key(topic);
        return false;
    }
    return true;
}
```

Listing 4.2: Key check function. The key struct is accessed from the hash table using the topic. If it is not present, and isn't currently waiting for response, a new key is requested and the check fails. Similarly expiry time and key length are checked. If all checks pass then a valid key is present.

Request Encryption Key. In the case where one of the key checks fails, a new encryption key must be requested from the KMS. The request is composed of the client's credentials, the topic and QoS level. The same security header format is used, except that the key ID field is replaced by the client ID. The other fields are encrypted with the private key and stored in the payload. This request is sent to the encryption key request topic, and the client waits for a response from the KMS.

Response Received. The client should always receive a response from the KMS, whether the request is authorised or not. If no response is received after a certain amount of time (network dependent), the request times out and the client sends another request. If the request has not been approved, a response is received with an error code corresponding to the reason for denial.

Decrypt AES-CCM. Upon receipt of response for an approved request, the message must be decrypted using the private key. This is achieved, as with all communications within the system, with AES-CCM encryption. This process should reveal the response, containing the key itself, its ID and time (milliseconds) until expiry.

Integrity Check. The first eight bytes of the decrypted message contain the MAC, which is used to make sure that the message is authentic and has not been tampered with in transmission. The plaintext message MAC is calculated and compared with the MAC received, and if they are equal then the message is accepted. If this check reveals that the message has been tampered with, a new key is requested and the offending message is destroyed.

Calculate Expiry Time. An obstacle that occurs in numerous embedded and resource-constrained device systems is clock synchronisation. The expiry time of the key on the device should match the expiry time stored by the KMS. It is assumed that the device does not have a synchronised clock. It does, however, have a kernel clock that is measured in upticks - number of milliseconds since the device started running. One of the fields within the response is the number of milliseconds until expiry. This should be added on to the uptick value at that point to get the uptick value when the key expires. This gives a good approximation to the expiry time stored in the KMS, however it does not account for the time taken for the packet to reach the device. In order to achieve a closer approximation, the KMS could measure the round-trip time to the device, halve it and add that on to the expiry time it had stored. In this case, it is not critical to have exact synchronisation between the two entities.

Populate Key Struct. Each time a new key is received, it is stored in a key struct according to the topic it is assigned to. This is implemented as a hash table, with the topic name as the key and key struct as the value. This process can either be creating a new entry, or replacing the values in the previous one. The struct contains the key, ID and time until expiry.

Message to Send. Once the key has been received, the system returns to the message that is to be sent, and the key check now indicates that a valid key exists.

Create Nonce. A random 8-byte nonce is used as IV for CCM encryption. It is important that the same nonce and key combination are never used more than once, as this leads to catastrophic failures and can reveal the plaintext in the case of an attack. For this reason a good entropy source should be used to seed the random number generator and ensure there is no repetition in nonce values.

Encrypt & Tag AES-CCM. With the nonce created and the key secured, the message the publisher wants to send can be encrypted. Using CCM mode, the MAC is calculated, and this is appended to the message before the whole sequence is encrypted.

Error Check. In the course of testing the encryption functionality, it was observed that in some cases the entire message would not get encrypted. The encryption function would truncate the original message, and a shorter encrypted message would be returned. Due to the properties of CCM encryption, the plaintext and ciphertext should both be the same length in successful encryption. A check is done once the message has been encrypted to make sure this is the case, and the encrypted message has not been truncated. If it has, a new nonce is created and encryption is repeated. A do-while loop ensures that the encrypted message is not sent until it is the same length as the original message.

Construct Payload. Once the message has been successfully encrypted, the payload can be constructed, with the security header, encrypted message and MAC. The security header consists of the fragmentation offset, key ID and nonce. The key ID and nonce are concatenated with the encrypted message and MAC, leaving the first byte of the security header for the fragmentation offset.

Fragmentation. The length of the payload is calculated and compared with the MTU of the system to identify how many fragments are needed. The message is then split up accordingly, using the first byte of each as the fragmentation offset. The final fragment sets the MSB of the offset to 1, indicating that no more fragments are to come.

```
unsigned char payload[PAYLOAD_SIZE];
unsigned char full_msg[FRAG_OFFSET_SIZE + KEY_ID_SIZE +
        NONCE_SIZE + sizeof(encrypted_msg)];
snprintf(full_msg, sizeof(full_msg), ``%c%c%s%s'', keyid_msb,
        keyid_lsb, nonce, encrypted_msg);

int num_fragments = sizeof(full_msg) / (PAYLOAD_SIZE-2);
if (sizeof(full_msg) % PAYLOAD_SIZE != 0)
        num_fragments++;

for (curr_frag = 1; curr_frag <= num_fragments; curr_frag++) {
    char fragment_offset = (char) curr_frag;
    if (curr_frag == num_fragments)
        fragment_offset |= 1UL << 7;
    snprintf(payload, sizeof(payload), ``%c%s'', fragment_offset,
    full_msg + ((PAYLOAD_SIZE-2)*(curr_frag -1)));
    publish_mqtt(payload, topic);
}
```

Listing 4.3: Payload construction and fragmentation implementation. The full message is constructed from the encrypted message and header fields. This length determines the number of fragments necessary. Each fragment concatenates the offset with the chunk of message which fits in the packet, according to the permitted payload size. This is then sent to the intended topic.

MQTT Send. Each fragment must be prepared as an MQTT packet to be sent to the broker. The fragment is put into the payload of the MQTT packet, and header information such as the topic name, QoS level and packet ID are added to the packet. This is then sent using MQTT infrastructure provided by the Zephyr Operating System. Due to congestion constraints with this MQTT implementation, a 500ms interval should be given between the sending of MQTT packets. When all fragments have been sent, the device can return to idle state to wait for another message to be sent.

4.5 IoT Service Module

The IoT service module provides support for resource-unconstrained entities within the system. These entities are clients within the MQTT framework, providing the IoT application processes. They generally provide some or all of the top four layers of the CISCO reference model (Fig. 2.1) - data accumulation, data abstraction, application and collaboration and processes. Data can be gathered and accumulated from any or all IoT devices in play, where it can be analysed and applied in a useful manner. The service can also present an interface through which to actuate a process or action on a device. This component is provided with a Python module that implements the same capabilities as the device library (Section 4.4). However, the implementation requires less overhead optimisation, due to the resource-unconstrained nature of the component. The decryption process is outlined, in the case where a message is received from a subscribed topic.

Decryption Process

When the client receives a message from the broker, it must be able to decrypt it in order to utilise the plaintext message appropriately. Fragmentation is handled so that the entire message is received and reassembled before decryption is carried out on it. Messages can be sent in one or several fragments, so for each fragment the MSB of the fragmentation offset is checked to see if the packet contains the final fragment. Sequential packet IDs indicate fragments that make up the same message, and differentiate between fragments from separate messages. When the first fragment of a message is received, the client parses the packet to extract the various fields. The fragmentation offset identifies that it is the first fragment, and therefore contains the security header. The nonce and MAC are stored, as well as the encrypted message payload, to be handled upon receipt of all fragments.

The key ID is obtained from the security header. It is stored as a binary value in two bytes in the header, and therefore is first converted to an integer. This value is used to check the key cache, which is a hash table with the key ID as the key and the key object as the value. In the case where the client is in possession of the required key, it is stored in the message object to be used once all fragments have been received. If not, the key is requested from the KMS, as per the process defined for the IoT device. This key request is performed asynchronously to the message fragment reception. With each fragment received, the encrypted message is concatenated. When the final fragment is obtained, a concurrent future method is used to decrypt the compiled message as soon as the required key is returned. Upon successful decryption, the application can make use of the plaintext method appropriately.

```
def decrypt(encrypted_msg, nonce, key):
    cipher = AES.new(key, AES.MODE_CCM, nonce)
    decrypted = cipher.decrypt(encrypted_msg)
    mac = decrypted[:8]
    msg = decrypted[8:]
    try:
        cipher.verify(mac)
        print "The message is authentic"
    except ValueError:
        print "Key incorrect or message corrupted"
        return None
    return msg
```

Listing 4.4: Decryption implementation. An 8-byte nonce and 16-byte key are used to decrypt messages. The first 8 bytes of the message represent the MAC, which is used to verify message integrity before the decrypted message is returned.

4.6 Summary

The design laid out in Chapter 3 has been implemented as an end-to-end IoT scheme, providing the required capabilities and services for all components of an MQTT-based IoT system. The broker requirements have been outlined, using a minimal-configuration Mosquitto implementation. The Key Management Service is executed as an autonomous resource for the system, running entirely over the MQTT protocol. It conforms to the decoupled and asynchronous nature of an MQTT system, adapting the request-response paradigm accordingly. Libraries equip the system components with the necessary requirements of the scheme, presenting an interface over which the processes outlined in the above sections can be carried out. The implementation has been constructed with the dual aims of maximising security and minimising overhead to the fullest extent possible.

Chapter 5

Evaluation

This chapter details a high-level overview of the system design and implementation, including experimental results of system performance, comparison with other security infrastructures, discussion and potential alternative solutions. The system is analysed under two lenses, the suitability for the MQTT-based IoT environment and the overall secureness of the system. A central focus of suitability for IoT is in minimising overhead - including bandwidth, time-to-idle, memory and the general overhead associated with system implementation. This is discussed in section 5.1. The security analysis of the scheme is contained within section 5.2. Finally, section 5.3 comprises an overall discussion of the approach, the reasoning behind design choices and an analysis of suitability.

5.1 Overhead

Minimising overhead on the IoT device was a fundamental objective in the design of the scheme. Several factors contributed to the realisation of this objective. First of all, a wide range of encryption techniques were analysed, and the most suitable algorithm was chosen with regard to minimising bandwidth and processing on the device. The design involves the smallest possible number of added messages to be sent. A light code footprint library was used to incur reduced memory requirements. The majority of security mechanisms exist externally to the device itself, only requiring the device to process the most crucial steps. In general, the implementation of the scheme is as lightweight as possible, and optimisations have been made to minimise overhead in all areas.

Bandwidth is minimised by ensuring all messages comprise as few bytes as possible. The first step in this endeavour is the choice of encryption technique. AES-128 CCM mode symmetric-key encryption was chosen. Symmetric-key encryption generally produces ciphertext that is the same length as the plaintext, without incurring any bloat.

This is in comparison to asymmetric-key techniques, which must increase the data size to obscure the length of the plaintext. This length information could be used along with the public key, to attempt potential plaintext sequences of such a length and compare them with the ciphertext. This is not an issue with private-key encryption, making it more efficient regarding bandwidth (McKay et al., 2017). Padding is, however, sometimes needed to ensure the length of the plaintext is as long as the block size of the symmetric algorithm. This is not necessary with CCM mode encryption.

Nineteen added bytes are required for every packet that is sent with the proposed scheme, compared to the same message with no security. These consist of the fragmentation offset (one byte), the key ID (two bytes), the nonce (eight bytes) and the MAC tag (8 bytes). Assuming an MTU of 127 bytes, this incurs a 15% overhead for a full packet. In the case of fragmentation, the full security header is not included in subsequent packets after the initial one, and only one extra byte for the fragmentation offset is required.

Comparing the bandwidth requirements to TLS, it can be assumed that the same ciphersuite is used, incurring a similar number of bytes for each message. The comparison is really in the key exchange policy, with TLS using a handshake for session establishment and the proposed scheme using MQTT requests and responses. TLS sets up a session for an initial message to be sent, and these sessions can be resumed with fewer requirements for subsequent messages. Considering the publish/subscribe framework, a TLS session must be established for each published message, and then the broker must establish a separate session for each subscriber to receive the message.

Message Field	**Average Length (Bytes)**
ClientHello	170
SessionID	32
ServerHello	75
Certificate	6000 (4×1500)
ClientKeyExchange	130
ChangeCipherSpec	1
Finished	12
TLS Record Header	5
TLS Handshake Header	4

Table 5.1: Average TLS handshake message sizes.

Table 5.1 outlines the average length of messages and message fields involved in the handshake procedure of TLS. An initial handshake sends seven handshake messages, four of those being record messages, and will incur approximately 6.5k bytes:

$$7 \times 4 + 4 \times 5 + 170 + 32 + 75 + 6000 + 130 + 2 \times 1 + 2 \times 12 = 6481$$

The overhead necessary to resume an existing TLS session is 332 bytes, using four handshake messages, three being record messages:

$$4 \times 4 + 3 \times 5 + 170 + 32 + 75 + 2 \times 1 + 2 \times 12 = 332$$

The initial handshake process must occur at least once per device, and the session can then be resumed at lesser cost for every message to be sent or received. This compares to the key request process of the proposed scheme. A request must be sent to the KMS when a new key for a topic is required, or if the old key expires. Expiration time for keys is variable depending on the system and network requirements, but can be an extended period of hours to days or weeks. The key request (both encryption and decryption) is of variable length, as it contains the client ID, password and topic name that all have variable lengths. It should not require fragmentation. Similarly, the response from the KMS contains the 16-byte key, expiration time and key ID, which will fit within one packet.

Parameter	**Value**
Publisher	1
Subscriber	1
Topic	1
Expiry Time	4 hours
Message	Every 10 minutes
Message Length	50 bytes
Request Length	60 bytes
Response Length	75 bytes
Experiment Length	1 day

Table 5.2: Overhead experiment parameters.

An experiment compares the overhead of the two systems, using parameters as outlined in table 5.2. This aims to model a simple IoT system, with one publisher publishing to a topic that is subscribed to by one subscriber. IoT systems generally send small messages (10 to 100 bytes) at regular intervals, which is reflected in this scenario. The parameters chosen are average values that are likely to occur in a real-world implementation of such a system. This experiment aims to give a general comparison between the two implementations, and it is noted that varying the parameters can create exponentially more complex systems that will give considerably different results.

TLS will require two session initialisations, between publisher and broker and then broker and subscriber. It will then require session resumption for each new message that is sent and received. This amounts to 108,578 bytes overall:

$$6481 + 6481 + 332 \times 2 \times (60 \div 10) \times 24 = 108,578$$

The proposed scheme will require an initial request and response for both the publisher and subscriber, and each time the key expires this is repeated. This amounts to 1,620 bytes:

$$(60 + 75) \times 2 \times (24 \div 4) = 1,620$$

There is evidently a huge bandwidth overhead involved in using TLS in comparison to the proposed scheme. In this common IoT scenario, the scheme incurs only 1.5% of the number of bytes necessary for the TLS implementation.

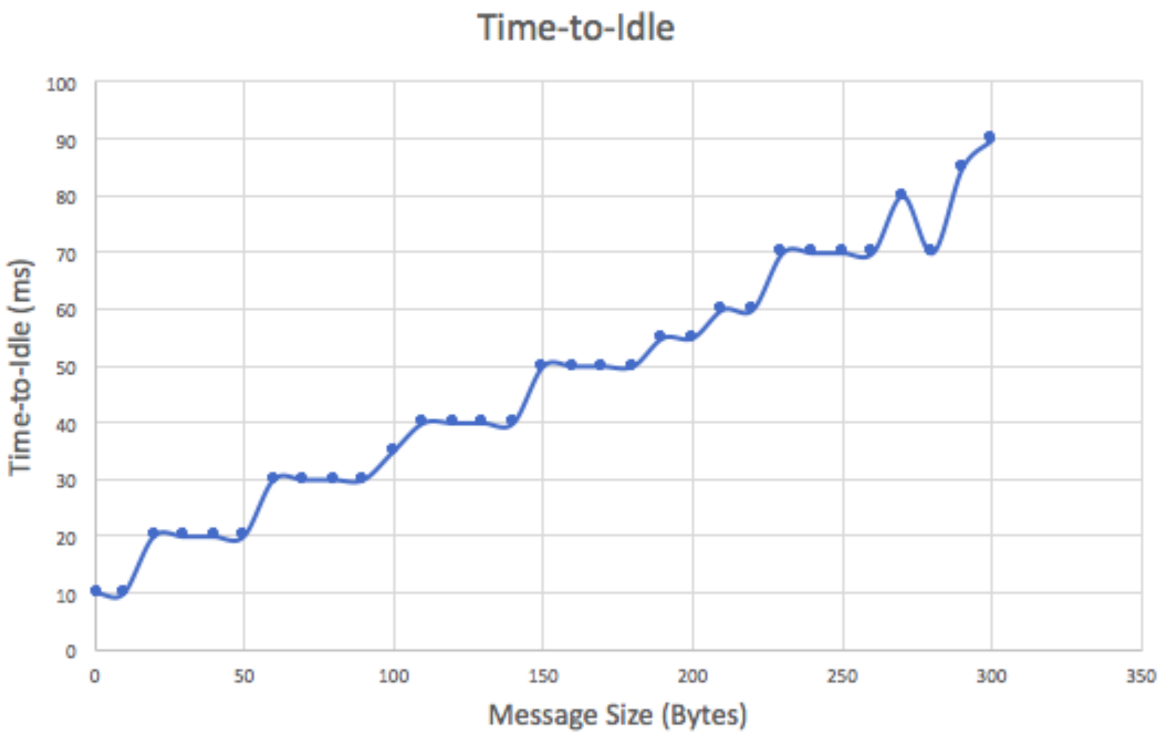

Figure 5.1: This graph analyses the average time the system takes from when a new message is to be sent, until it is encrypted, sent and the device returns to idle state. This varies with message size. These measurements assume the key is cached locally.

IoT devices aim to exist in idle state for as long as possible, and to return to this idle state quickly after an event has been processed. This is crucial for reserving resources, and should be a consideration in the system design by way of minimising processing time. Fig. 5.1 presents the time taken for the device to return to idle state once a message of a certain size is encrypted and sent. These measurements have been taken with the required key stored in the device's local cache. The graph exhibits a step-wise linear relationship, with the time increasing relative to the size of the message to be sent. The message size is the deciding factor in the time incrementation, as it relates to the number of blocks to be processed by the encryption algorithm. Fragmentation occurs at the MTU size of 127 bytes, yet the graph shows no obvious jump in processing time at these points where multiple messages are sent.

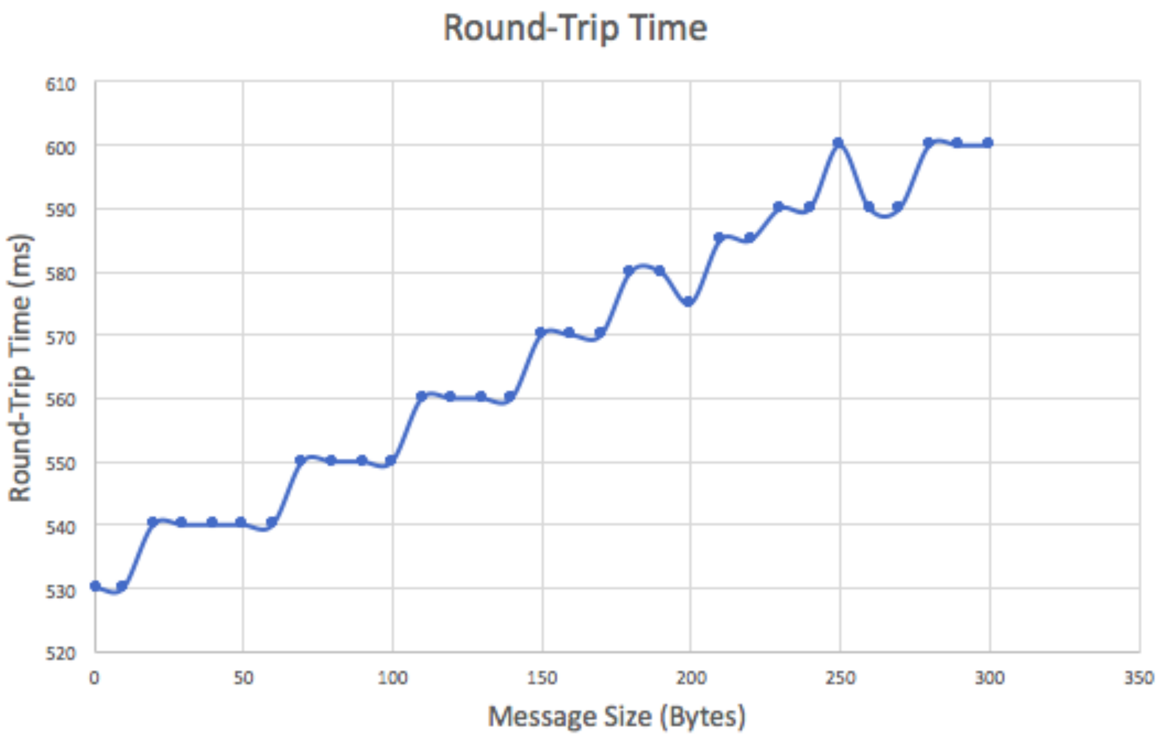

Figure 5.2: This graph analyses the average time the system takes from when a new message is to be sent, until the key is requested and received, the message is encrypted, sent and the device returns to idle state. This varies with message size.

The graph in Fig. 5.1 assumes the required key is stored locally. Round-trip time measurements have also been collected for the case where a key request must be made to the KMS before a message is to be sent. This can be seen in Fig. 5.2. This portrays an extremely similar step-wise linear relationship, though with an upward shift on the time axis. This indicates that the request time is a constant (approximately 520ms), which gets added on to the message processing time once the key has been retrieved. In both graphs it can be seen that there is slightly more variation in the processing time for larger messages. This is due to the encryption error that sometimes occurs - when the full message is not encrypted and encryption must be repeated until the correctly-sized ciphertext is produced. This is more likely to occur in larger messages, and therefore affects the average processing time for such messages.

The times taken for messages to be processed and sent by the scheme are negligible for most real-world IoT applications. A 40ms latency in transmitting a full packet will not negatively impact the operation of a system. In particular, for this configuration, a 500ms delay must be implemented between any MQTT messages that are sent. This is because after a short amount of time of consistent MQTT messages, the transmission mechanism becomes flooded and ceases to operate. As a result of this, the throughput for the scheme and for the system with no security implementation is extremely similar. For the most part, the security processing can be done within the required delay timeframe. In the occasional case where a new key is required, the delay will be slightly longer than the required delay. This is in comparison to TLS, which must account for the round-trip time of four handshake messages whenever a session is resumed for a new message to be sent.

The general implementation overhead of the system is relatively low, particularly in comparison with TLS. It is clear that in the resource-constrained environment of IoT, TLS can be wholly unsuitable, requiring a considerable amount of extra bandwidth and processing than is necessary for an IoT message that, relatively, is extremely small. The proposed solution offers a manageable security scheme, designed specifically for such an environment. The KMS manages the bulk of the computation, allowing the IoT devices to operate with minimal awareness of the security infrastructure in place.

5.2 Secureness

The fundamental purpose of the design is to provide a secure means of communication between IoT devices and the system components, through the MQTT broker. It must be ensured that the security is robust, and the service being provided by the scheme is legitimate. By definition of a security system, it should protect its client against threats and vulnerabilities. The end-user must trust the scheme to fully and reliably maintain this purpose. Risks can be very high in the case of failure. For these reasons, a security scheme such as this must undergo rigorous testing to ensure end-to-end secureness. This section outlines a slightly more lightweight security evaluation, as this is a first iteration of the scheme. Upon further development, more rigorous real-world penetration testing can be carried out to identify any vulnerabilities that should be addressed. However, the confidence in the system at this level of analysis is high.

Complexity is the downfall of a security system (Bellovin, 2016). An overly-complex system can lead to the appearance of security flaws that are difficult to identify and analyse. When designing the scheme, simplicity was an essential objective - both in terms of security and overhead minimisation. The core security components are the encryption and MAC implementation, key caching and Key Management Service. Within the KMS are security aspects such as authentication, authorisation, key exchange and system monitoring. These all provide essential, yet simple, functionalities to the secure and robust communication between IoT clients within the system. The components must work autonomously and in tandem to provide the end-to-end security expected. The failure of any one aspect can jeopardise the entire system. In order to analyse the security, potential threats and attacks will be considered. This will reveal how well the system defends against such an attack.

The first attack to consider is that of a compromised broker. IoT systems often use an open, third-party broker to route messages between clients. With no security implementation in place, an attacker can access such a broker, subscribe to a topic and receive any published messages. It could also launch an attack on the broker with the aim to access any messages presently stored within it. The proposed scheme adequately deals with such an attack. Firstly, the scheme requires all devices to be authenticated by the KMS and the broker, and any node whose ID and password is not on the whitelist will not be permitted to subscribe to a topic. On top of this, any message that is stored in the broker is encrypted, therefore a broker attack will not reveal any sensitive information. The broker is a separate entity with no access to or information on the keys needed to decrypt the messages.

Some common attacks to consider within TCP-based communications are eavesdropping attacks and Man in the Middle attacks (Tuna et al., 2017). An eavesdropping attack is when packets are sniffed in transit, and the data within them is gathered. The payload will be encrypted, and thus an eavesdropping attack will not reveal any sensitive information. As outlined in Section 3.4, AES-128 is secure against attacks in which the attacker has no access to the required key. A Man in the Middle attack similarly occurs when packets are intercepted in transmission, and are modified before being forwarded on to the broker or subscriber. This attack targets the integrity of the message, which is secured with the Message Authentication Code (MAC) included within CCM encryption. The recipient will perform an integrity check on the message, which will reveal whether the message has been tampered with in any way.

IoT systems can often be vulnerable to Distributed Denial of Service (DDoS) botnet attacks. A botnet harnesses a number of unsecured devices through implanted malware, and uses them to launch an extensive DDoS attack (Jerkins, 2017). This entails using the devices to send a large amount of requests, flooding a target system and rendering it unusable. These devices are vulnerable because of their lack of security, but due to the authentication and authorisation mechanisms within the scheme, only approved communication can occur and botnets cannot penetrate such a network. On the other hand, the IoT system itself can be vulnerable to a DoS attack. The KMS performs passive traffic analysis on the overall system, identifying patterns and isolating any requests likely to overload the system. The KMS can then blacklist such clients immediately, and alert the system administrator to mitigate the damage.

In order to adequately evaluate the security of the scheme, it must be compared to the security of a TLS implementation. The immediate advantage of the scheme over TLS is the provision of end-to-end security, whereas TLS only provides security from broker to client. TLS maintains an impenetrable channel between sender and receiver over which encrypted messages can be sent. This scheme does not provide such a channel, relying only on the encryption mechanisms for defense. The lack of private channel allows packets to be intercepted, though as detailed above, no sensitive information will be revealed. However, the header fields exist in plaintext, and an attacker can glean some information, such as topic name, by accessing packets and interpreting the header information. An attacker can gather information on the regular patterns of communication, such as timing, regularity and topic, and use this to launch a more sinister attack. This cannot occur when an established channel exists between entities, as in a TLS implementation.

Due to the fact that payload encryption exists on the application layer and TLS exists below this layer, the proposed scheme can be utilised in tandem with TLS. This will provide both secure channel and end-to-end security for an application. The scheme has been designed with the perspective of an extremely resource-constrained IoT system. This is not always the case in IoT, and many devices have enough compute power to support protocols such as TLS. It is still important to design a system for efficiency, and optimise for processing and bandwidth constraints. IoT applications with fewer constraints will still have use for such a system, and payload encryption in conjunction with TLS will provide increased security.

5.3 Discussion

As outlined in Sections 5.1 and 5.2, the scheme successfully provides authenticated payload encryption to a resource-constrained environment such as IoT. It affords an alternative to TLS for the MQTT framework, improving upon TLS in several respects. This is particularly important for real-world IoT applications that cannot support resource-intensive security mechanisms. This leads to the deployment of insecure systems, due to the lack of security options available. The scheme aims to be adaptable to a wide range of applications, providing capabilities through libraries for a straightforward implementation. It is designed to be flexible in accommodating applications with a wide range of requirements, capabilities and environments. The scheme is scalable, with no limit to the number of components that can be supported. The only limit to scalability is in the initial deployment, in which a new device must be registered with the KMS. Once this step is complete, the security scheme will work autonomously as required by the system.

Several overarching decisions were made upon the inception and continued development of the scheme, in order to adequately provide a solution for the proposed problem. The initial decision was to examine MQTT as an IoT protocol, and provide an applied security solution to suit such a protocol and environment. The choice was made to focus on MQTT over a number of other protocols, such as CoAP, AMQP, XMPP or MQTT-SN. A symmetric-key implementation was selected, over TLS or asymmetric-key encryption. The design was driven by the objective of providing a lightweight security implementation, and design decisions were formed based on balancing security and usability.

MQTT is the most widely used and accepted application-layer protocol in practical IoT applications. Its publish/subscribe architecture is suitable for many IoT scenarios, though due to the fact that it runs over TCP/IP, it is not the most lightweight framework. It has little to no security infrastructure, recommending TLS as a security layer, which is generally too resource-intensive to be supported in IoT scenarios. MQTT-SN (MQTT for Sensor Networks) was also considered. This is an adapted protocol that does not require the TCP/IP stack, and optimises MQTT for low-powered networks (Zhao & Ge, 2013). However, MQTT-SN is not well supported, and is not used extensively in IoT. The design aims to improve the current state of IoT by providing a suitable security scheme for the most widely used IoT protocol.

The decision was made to devise the entire system through the same protocol - MQTT. This involves all components communicating through the broker, including the KMS. Another option that was considered was to use a different protocol, such as CoAP or HTTP, for RESTful communications with the KMS for key exchange. This would essentially provide the KMS functionalities as an entirely separate system to the already established MQTT system. CoAP is a suitable protocol for such a scenario, due to its lightweight request/response architecture. However, it was deemed that the added overhead involved in supporting another protocol was unnecessary. As demonstrated, MQTT can be adequately adapted to fit a request/response framework. Using one protocol presents the added benefit of compliance and flexibility with the needs of the application. It also allows straightforward system monitoring, and the ability to synchronise with the broker.

Some design decisions have been made that given more time and development, could be altered and implemented in a more efficient fashion. For example, as a result of the constraints of the development system used, it was necessary to implement fragmentation on the application layer. This is due to the fact that messages over the MTU length would not send, and fragmentation was not realised on a lower layer. 6LoWPAN guarantees fragmentation in order to adapt large IPv6 packets to size requirements of constrained protocols (Wang & Mu, 2017). This was not enforced in the 6LoWPAN implementation used, and therefore it was necessary to devise a fragmentation policy. Upon further development, 6LoWPAN fragmentation should be utilised.

Another design decision that could be reconsidered or analysed further is the cache update policy. The scheme currently updates the key cache only when necessary, i.e. when a message is to be sent and it is found that an invalid key exists in the cache. A more efficient method could be enforced that will trigger an interrupt when the system recognises that a key in the cache has become invalid, usually by expiring. Both implementations have merit, and it may depend on the requirements of the application to define the correct strategy to use. The current strategy allows the system to only request keys that it needs, without making any unnecessary requests to the KMS. The alternative strategy allows keys to be constantly up-to-date, and there will be no added delay in messages that must wait for a key response before sending. The decision comes down to prioritising low message latency or minimal key requests. It can also vary depending on the number and variation of topics being used by the same device.

In order to improve upon TLS for resource-constrained environments, the chosen encryption must minimise processing, memory requirements and added overhead. Symmetric-key encryption has a relatively low computational overhead compared to asymmetric key encryption, requires lower bandwidth and has no need for storing large keying materials in memory. AES-128 CCM mode encryption was chosen due to its provision of encryption and message integrity, its extremely low bandwidth requirements, and its strength of security. Hybrid approaches use symmetric techniques for encryption and asymmetric techniques for key exchange. The design improves upon these approaches by offsetting the key management processing to a resource-unconstrained service. Overall this leads to a secure, reliable low-overhead system.

5.4 Summary

The scheme has been evaluated through examining how secure of a system it is, as well as how well it suits the IoT paradigm. These are the two pillars that the scheme has been designed and built around from conception. All of the major design decisions have been formulated based on these criteria. The overhead incurred by the system has been analysed, and compared with the de-facto standard security protocol, TLS. This analysis shows how significantly it improves upon TLS with regard to bandwidth, processing and time-to-idle. In Section 5.2 the security of the scheme is evaluated, considering the system components, potential security breaches and a TLS comparison. It was found that the system is secure against many common attacks, as it adequately provides the CIA triad - confidentiality, integrity and availability. Finally, design decisions were scrutinised to assess their suitability and validity for the scheme.

Chapter 6

Conclusions & Future Work

6.1 Conclusions

An authenticated payload encryption scheme has been developed for use in MQTT-based IoT systems. The area was chosen to focus on due to the exponential growth of the IoT industry, and the lack of appropriate security solutions available for such resource-constrained environments. The solution aims to provide a flexible, lightweight end-to-end security scheme for the publish/subscribe architecture of MQTT. All essential security facets have been provided for, resulting in a complete security architecture that can be integrated seamlessly into a real-world IoT application. This compares to several solutions that only focus on one aspect of security, such as encryption (Katsikeas et al., 2017) or authentication (Moustaine & Laurent, 2012).

The design was heavily optimised to suit a resource-constrained environment, and this lead to an efficient system. A significant amount of research and experimentation were carried out to find the ideal techniques and algorithms to use, in order to balance security with usability. The purpose of the scheme is primarily to provide security, and this was always at the forefront of decision-making. Any overhead optimisations that could jeopardise the robust security of the scheme were eliminated. Security centres around trust, and a security scheme must guarantee the trustworthiness and legitimacy of its security implementation to its users. Nonetheless, a lightweight protocol was still developed with significant improvements over TLS in bandwidth, average packet size, memory, computation and time-to-idle. The scheme successfully fulfills its objectives and provides a solution to the formulated problem.

6.2 Future Work

The scheme has been developed as a functional and usable system, as outlined in Chapter 4. Upon further development, the system could be optimised and tested more rigorously. Each aspect of the scheme could be thoroughly analysed considering the most efficient use of resources, and alternative design decisions could be compared and tested to prove or disprove their effectiveness. In terms of security, penetration testing can be applied to the system to identify any security vulnerabilities that should be dealt with and improved upon. This would involve launching a range of simulated attacks on the system, in order to evaluate its strength of security. Finally, the scheme could be applied to a real-world IoT system, in order to evaluate its functional overhead and usability, and assess the theoretical grounds it was built upon. As it stands, the scheme operates satisfactorily, and with further testing and development, a more robust and durable system could be produced.

Bibliography

A. Banks, R. G. (2014). Mqtt version 3.1.1. *OASIS Standard*.

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. volume 17 (pp. 2347–2376).

Alaba, F. A., Othman, M., Hashem, I. A. T., & Alotaibi, F. (2017). Internet of things security: A survey. *Journal of Network and Computer Applications*, 88, 10 – 28.

Almuhammadi, S. & Al-Hejri, I. (2017). A comparative analysis of aes common modes of operation. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)* (pp. 1–4).

Ammar, M., Russello, G., & Crispo, B. (2018). Internet of things: A survey on the security of iot frameworks. *Journal of Information Security and Applications*, 38, 8 – 27.

Bacon, J., Eyers, D., & Singh, J. (2009). Security in multi-domain event-based systems. *it - Information Technology*, 51(5), 277.

Bellovin, S. M. (2016). *Thinking Security: Stopping next Year's Hackers*. Addison-Wesley.

Chung, J. H. & Cho, T. H. (2016). Adaptive energy-efficient ssl/tls method using fuzzy logic for the mqtt-based internet of things. *International Journal Of Engineering And Computer Science*, 5(12).

CISCO (2014). Cisco iot reference model white paper. `http://cdn.iotwf.com/resources/71/IoT_Reference_Model_White_Paper_June_4_2014.pdf`. Accessed: 2018-03-30.

Darroudi, S. M. & Gomez, C. (2017). Bluetooth low energy mesh networks: A survey. *Sensors (14248220)*, 17(7), 1 – 19.

Fortify, H. P. (2014). Hp internet of things research study. *Hewlett Packard Enterprise*.

Hoeppe, A. (2017). Predicts 2017: Industrie 4.0. *Gartner.*

Jan, M. A., Khan, F., Alam, M., & Usman, M. (2017). A payload-based mutual authentication scheme for internet of things. *Future Generation Computer Systems.*

Jerkins, J. A. (2017). Motivating a market or regulatory solution to iot insecurity with the mirai botnet code. In *Computing and Communication Workshop and Conference (CCWC), 2017 IEEE 7th Annual* (pp. 1–5).: IEEE.

Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F., & Alonso-Zarate, J. (2015). A survey on application layer protocols for the internet of things. *Transaction on IoT and Cloud Computing*, 3(1), 11–17.

Katsikeas, S., Fysarakis, K., & Miaoudakis, A. (2017). Lightweight & secure industrial iot communications via the mq telemetry transport protocol. *2017 IEEE Symposium on Computers and Communications (ISCC), Computers and Communications (ISCC), 2017 IEEE Symposium on*, (pp. 1193).

Katz, J. (2014). *Introduction to modern cryptography.* Chapman & Hall/CRC.

Kong, J. H., Ang, L.-M., & Seng, K. P. (2015). A comprehensive survey of modern symmetric cryptographic solutions for resource constrained environments. *Journal of Network and Computer Applications*, 49, 15 – 50.

Kouicem, D. E., Bouabdallah, A., & Lakhlef, H. (2018). Internet of things security: A top-down survey. *Computer Networks.*

McKay, K. A., Feldman, L., & Witte, G. A. (2017). Report on lightweight cryptography.

Mektoubi, A., Hassani, H. L., Belhadaoui, H., Rifi, M., & Zakari, A. (2016). New approach for securing communication over mqtt protocol a comparaison between rsa and elliptic curve. In *Systems of Collaboration (SysCo), International Conference on* (pp. 1–6).: IEEE.

Morabito, R., Cozzolino, V., Ding, A. Y., Beijar, N., & Ott, J. (2018). Consolidate iot edge computing with lightweight virtualization. *IEEE Network*, 32(1), 102–111.

Mosenia, A. & Jha, N. K. (2017). A comprehensive study of security of internet-of-things. *IEEE Transactions on Emerging Topics in Computing*, 5(4), 586–602.

Moustaine, E. E. & Laurent, M. (2012). A lattice based authentication for low-cost rfid. In *2012 IEEE International Conference on RFID-Technologies and Applications (RFID-TA)* (pp. 68–73).

Neisse, R., Steri, G., & Baldini, G. (2014). Enforcement of security policy rules for the internet of things. In *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)* (pp. 165–172).

Neisse, R., Steri, G., Fovino, I. N., & Baldini, G. (2015). Seckit: A model-based security toolkit for the internet of things. *Computers & Security*, 54(Secure Information Reuse and Integration & Availability, Reliability and Security 2014), 60 – 76.

Nguyen, K. T., Laurent, M., & Oualha, N. (2015). Survey on secure communication protocols for the internet of things. *Ad Hoc Networks*, 32, 17 – 31. Internet of Things security and privacy: design methods and optimization.

Onica, E., Felber, P., Mercier, H., & Riviere, E. (2016). Confidentiality-preserving publish/subscribe, a survey. *ACM Computing Surveys*, 49(2), 1–43.

Rizzardi, A., Sicari, S., Miorandi, D., & Coen-Porisini, A. (2016). Aups: An open source authenticated publish/subscribe system for the internet of things. *Information Systems*, 62, 29 – 41.

Shin, S., Kobara, K., Chuang, C.-C., & Huang, W. (2016). A security framework for mqtt. In *2016 IEEE Conference on Communications and Network Security (CNS)* (pp. 432–436).

Sicari, S., Rizzardi, A., Grieco, L., & Coen-Porisini, A. (2015). Security, privacy and trust in internet of things: The road ahead. *Computer Networks*, 76, 146 – 164.

Singh, M., Rajan, M. A., Shivraj, V. L., & Balamuralidhar, P. (2015). Secure mqtt for internet of things (iot). In *2015 Fifth International Conference on Communication Systems and Network Technologies* (pp. 746–751).

Szalachowski, P., Ksiezopolski, B., & Kotulski, Z. (2010). Cmac, ccm and gcm/gmac: Advanced modes of operation of symmetric block ciphers in wireless sensor networks. *Information Processing Letters*, 110(7), 247 – 251.

T. Dierks, E. R. (2008). The transport layer security (tls) protocol, version 1.2. *IETF RFC 5246*.

Tuna, G., Kogias, D. G., Gungor, V. C., Gezer, C., Takn, E., & Ayday, E. (2017). A survey on information security threats and solutions for machine to machine (m2m) communications. *Journal of Parallel and Distributed Computing*, 109, 142 – 154.

Unger, S., Pfeiffer, S., & Timmermann, D. (2012). Dethroning transport layer security in the embedded world. In *2012 5th International Conference on New Technologies, Mobility and Security (NTMS)* (pp. 1–5).

Urien, P. (2017). Introducing tls/dtls secure access modules for iot frameworks: Concepts and experiments. In *2017 IEEE Symposium on Computers and Communications (ISCC)* (pp. 220–227).

Wang, X. & Mu, Y. (2017). Communication security and privacy support in 6lowpan. *Journal of Information Security and Applications*, 34, 108 – 119.

Yassein, M. B., Shatnawi, M. Q., Aljwarneh, S., & Al-Hatmi, R. (2017). Internet of things: Survey and open issues of mqtt protocol. In *2017 International Conference on Engineering MIS (ICEMIS)* (pp. 1–6).

Z. Shelby, K. Hartke, C. B. (2014). The constrained application protocol (coap). *Internet Engineering Task Force (IETF).*

Zhao, K. & Ge, L. (2013). A survey on the internet of things security. In *Computational Intelligence and Security (CIS), 2013 9th International Conference on* (pp. 663–667).: IEEE.