

# Securing Communication between ESP32-based IOT Devices and Remote Servers: An Analysis of SSL Certificate Verification Methods

Gopikrishnan S

Student

Hindusthan Institute of Technology

**Abstract-** IoT devices are commonly connected with remote servers / cloud servers through only HTTP (Hyper Text Transfer Protocol) in a plain text mode which means the data send and receive through this communication is more vulnerable. It can be read by any one. There are ongoing researches to fix this by using HTTPS (Hypertext Transfer Protocol Secure) communication which uses (SSL/TLS) protocols to ensure data security on remote communications. There are some improvements on the researches but the full solution is not found yet. They are not fully supporting the ESP32 types on installation and implementation.

The research explores HTTPS libraries used for encrypted communication using SSL/TLS certificate. The research finds a way to achieve a secure communication (HTTPS) between an IoT device (arduino ESP32) and a remote/cloud Node js and MQTT server through installing a SSL/TLS protocol on client IoT device. A JSON library is used to parse and format data transferred to the cloud server from a IOT device. Esp32 dev-kit is used in this experiment. The IoT device is used to provide the data to the weather monitoring system of a solar plant.

## CHAPTER 1 INTRODUCTION

### 1.1 OVERVIEW

The growing trend of Internet of Things (IoT) devices has led to an increased need for secure communication between these devices and remote servers. The ESP32 microcontroller is a popular choice for IoT devices due to its low cost and wireless capabilities. However, ensuring the security of communication between ESP32-based devices and remote servers presents several challenges, particularly with regards to SSL certificate verification. In this research paper, we analyze various SSL certificate verification methods and evaluate their effectiveness in securing communication between ESP32-based IoT devices and remote servers. The results of our analysis provide insights into the strengths and weaknesses of these methods and offer recommendations for implementing secure communication in ESP32-based IoT systems.

IoT devices are the nonstandard computing devices that connect wirelessly to a network and have the ability to transmit data, such as the many devices on the internet of things (IoT).

IoT network protocols are used to connect devices over the network. These are the set of communication protocols typically used over the Internet. Using IoT network protocols, end-to-end data communication within the scope of the network is allowed.

HyperText Transfer Protocol (HTTP) is the best example of IoT network protocol. This protocol has formed the foundation of data communication over the web. It is the most common protocol that is used for IoT devices when there is a lot of data to be published. However, the HTTP protocol is not preferred because of its cost, battery-life, energy saving, and more constraints.

The main problem here is the data vulnerability. Because HTTP requests and responses are sent in plain text, which means that anyone can read them. To avoid this situation here we should use some secure communication protocols like Secure Sockets Layer (SSL) / Transport Layer Security (TLS).

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP). It is used for secure communication over a computer network, and is widely used on the Internet. In HTTPS, the communication protocol is encrypted using Transport Layer Security (TLS) or, formerly, Secure Sockets Layer (SSL). The protocol is therefore also referred to as HTTP over TLS, or HTTP over SSL.

MQTT is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth.

A **cloud server** is a pooled, centralized server resource that is hosted and delivered over a network—typically the Internet—and accessed on demand by multiple users. Cloud servers can perform all the same functions of a traditional physical server, delivering processing power, storage and applications.

A cloud server is made possible through virtualization. Management software called a hypervisor is installed on physical servers to connect and virtualize them: abstracting their combined resources and pooling them together to create virtual servers. These virtual resources can then be automated and delivered over the cloud for shared use in a single organization or across multiple organizations.

Node.js is an open-source server environment. Node.js is cross-platform and runs on Windows, Linux, Unix, and macOS. Node.js is a back-end JavaScript runtime environment. Node.js runs on the V8 JavaScript Engine and executes JavaScript code outside a web browser.

Node.js lets developers use JavaScript to write command line tools and for server-side scripting. The functionality of running scripts server-side produces dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js represents a "JavaScript everywhere" paradigm, unifying web-application development around a single programming language, rather than different languages for server-side and client-side scripts.

Node.js has an event-driven architecture capable of asynchronous I/O. These design choices aim to optimize throughput and scalability in web applications with many input/output operations, as well as for real-time Web applications (e.g., real-time communication programs and browser games).

The Node.js distributed development project was previously governed by the Node.js Foundation, and has now merged with the JS Foundation to form the OpenJS Foundation. OpenJS Foundation is facilitated by the Linux Foundation's Collaborative Projects program.

## 1.2 OBJECTIVES

- To evaluate the security of the SSL/TLS communication between ESP32-based IoT devices and remote servers.
- To investigate the different SSL certificate verification methods used in ESP32-based IoT devices and to identify their strengths and weaknesses.
- To analyze the feasibility of implementing SSL certificate verification methods in ESP32-based IoT devices with limited resources and computational capabilities.
- To propose a comprehensive framework for securing the communication between ESP32-based IoT devices and remote servers by integrating the most effective SSL certificate verification methods.
- To evaluate the performance and effectiveness of the proposed framework through empirical experiments and compare it with existing solutions.

## CHAPTER 2 LITERATURE SURVEY

### [1.] Pilny Tomas “WiFiClientSecure on Arduino-Esp32”, 2021 .

The WiFiClientSecure class implements support for secure connections using TLS (SSL). It inherits from WiFiClient and thus implements a superset of that class' interface. There are three ways to establish a secure connection using the WiFiClientSecure class: using a root certificate authority (CA) cert, using a root CA cert plus a client cert and key, and using a pre-shared key (PSK).

### [2.] Dan Dragomir, Laura Gheorghe, Research and Development Department, Academy of Romanian Scientists Bucharest, Romania

### ,Sergiu Costea, Alexandru Radovici “A Survey on Secure Communication Protocols for IoT Systems”, 2016 .

The Internet of Things (IoT) integrates a large number of physical objects that are uniquely identified, ubiquitously interconnected and accessible through the Internet. IoT aims to transform any object in the real-world into a computing device that has sensing, communication and control capabilities. There is a growing number of IoT devices and applications and this leads to an increase in the number and complexity of malicious attacks. It is important to protect IoT systems against malicious attacks, especially to prevent attackers from obtaining control over the devices. A large number of security research solutions for IoT have been proposed in the last years, but most of them are not standardized or interoperable. In this paper, we investigate the security capabilities of existing protocols and networking stacks for IoT. We focus on solutions specified by well-known standardization bodies such as IEEE and IETF, and industry alliances, such as NFC Forum, ZigBee Alliance, Thread Group and LoRa Alliance.

### [3.] Kim Thuat Nguyena, Maryline Laurentb, NouhaOualhaa “Survey on secure communication protocols for the Internet of Things”, 2015

The Internet of Things or “IoT” defines a highly interconnected network of heterogeneous devices where all kinds of communications seem to be possible, even unauthorized ones. As a result, the security requirement for such network becomes critical whilst common standard Internet security protocols are recognized as unusable in this type of networks, particularly due to some classes of IoT devices with constrained resources. The document discusses the applicability and limitations of existing IP-based Internet security protocols and other security protocols used in wireless sensor networks, which are potentially suitable in the context of IoT. The analysis of these protocols is discussed based on a taxonomy focusing on the key distribution mechanism.

**[4.] Fagen Li, Pan Xiong “Practical Secure Communication for Integrating Wireless Sensor Networks Into the Internet of Things”, 2013**

If a wireless sensor network (WSN) is integrated into the Internet as a part of the Internet of things (IoT), there will appear new security challenges, such as setup of a secure channel between a sensor node and an Internet host. In this paper, we propose a heterogeneous online and offline signcryption scheme to secure communication between a sensor node and an Internet host. We prove that this scheme is indistinguishable against adaptive chosen ciphertext attacks under the bilinear Diffie-Hellman inversion problem and existential unforgeability against adaptive chosen messages attacks under the  $q$ -strong Diffie-Hellman problem in the random oracle model. Our scheme has the following advantages. First, it achieves confidentiality, integrity, authentication, and non-repudiation in a logical single step. Second, it allows a sensor node in an identity-based cryptography to send a message to an Internet host in a public key infrastructure. Third, it splits the signcryption into two phases: i) offline phase; and ii) online phase. In the offline phase, most heavy computations are done without the knowledge of a message. In the online phase, only light computations are done when a message is available. Our scheme is very suitable to provide security solution for integrating WSN into the IoT.

### CHAPTER 3 SYSTEM ANALYSIS

#### 3.1 EXISTING SYSTEM

##### HTTP with IoT

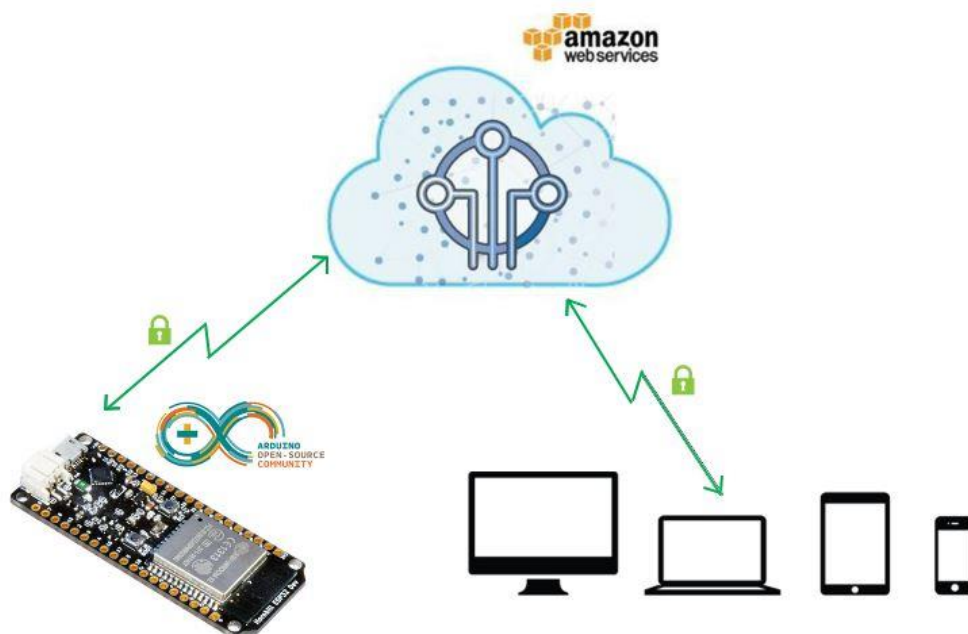
HyperText Transfer Protocol is the best example of IoT network protocol. This protocol has formed the foundation of data communication over the web. It is the most common protocol that is used for IoT devices when there is a lot of data to be published. However, the HTTP protocol is not preferred because of its cost, battery-life, energy saving, and more constraints.

The point to understand is that HTTP transfer data as plain text, which means the data can be read by anyone. So the data vulnerability occurs. And there is no data security.

#### 3.2 PROPOSED APPROACH

- **Using HTTPS with IoT**

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP). It is used for secure communication over a computer network, and is widely used on the Internet. In HTTPS, the communication protocol is encrypted using Transport Layer Security (TLS) or, formerly, Secure Sockets Layer (SSL). The protocol is therefore also referred to as HTTP over TLS, or HTTP over SSL.



HTTPS here ensures the secure communication between the IoT device and the remote node js cloud server. The SSL / TLS here encrypts the data we post to server and the response we get back from server. So it ensures data security.

Fig. 3.2 Using HTTPS with IoT

##### 3.2.1 Asymmetric Encryption

This involves two keys: a public key and a private key. Generally, public keys are freely distributed, private keys are kept, well, private. These two keys are mathematically related. While we won't get into the mathematical complexities, here's what we should know:

- Data encrypted by the public key can be decrypted by the private key
- A private key can digitally sign documents, and a public key can verify if the digital signature is valid and confirm that the document hasn't been altered. In addition to verifying the integrity of a document, a digital signature also proves the ownership of the private key corresponding to the public key. Only the owner of the corresponding private key could have signed the document whose validity was determined by the public key we have. This is an extremely important point.

### 3.2.2 Symmetric Encryption

- This is a simple form of encryption, using a single key. The same key that is used for encrypting the data can also be used for decrypting the data.
- Symmetric encryption is faster than asymmetric encryption for two major reasons:
- It uses much shorter keys (128 or 256-bit keys are common) than asymmetric encryption (where 1024 and 2048-bit are common key lengths)
- It uses simpler mathematical operations like multiplication or XOR in the process of encryption whereas asymmetric encryption uses complex operations like power and modulus.

### 3.2.3 Symmetric vs Asymmetric:

Both symmetric and asymmetric encryption have their advantages and disadvantages, and the choice between them depends on the specific use case and security requirements.

Symmetric encryption uses the same key for both encryption and decryption, which makes it fast and efficient. It is well suited for encrypting large amounts of data, such as in data storage or communication protocols. However, the challenge with symmetric encryption is how to securely share the key between the parties involved in the communication. If the key is compromised, then the encrypted data can be decrypted and read.

Asymmetric encryption, on the other hand, uses a pair of keys - a public key for encryption and a private key for decryption. This makes it well suited for authentication, digital signatures, and key exchange protocols. Asymmetric encryption does not require the parties to share a secret key, which eliminates the key distribution problem of symmetric encryption. However, asymmetric encryption can be much slower than symmetric encryption, and it is not well suited for encrypting large amounts of data.

In summary, symmetric encryption is better suited for encrypting large amounts of data, while asymmetric encryption is better suited for authentication, digital signatures, and key exchange protocols. Both types of encryption have their own strengths and weaknesses, and the choice between them depends on the specific use case and security requirements.

## 3.3 ADVANTAGES

- HTTPS uses SSL that provides the encryption of the data.
- It is secure as it sends the encrypted data which hackers cannot understand.
- Ensures data security while the remote communication takes place between remote server /cloud server and IoT device (ESP32).
- Leveraged to gain customer trust.
- Gives protection against Phishing.

## 3.4 FEASIBILITY STUDY

Feasibility studies aim to objectively and rationally uncover the strengths and weaknesses of the existing business or proposed venture, opportunities and threats as presented by the environment, the resources required to carry through, and ultimately the prospects for success. In its simplest term, the two criteria to judge feasibility are cost required and value to be attained. As such, a well-designed feasibility study should provide a historical background of the business or project, description of the product or service, accounting statements, details of the operations and management, marketing research and policies, financial data, legal requirements and tax obligations. Generally, feasibility studies precede technical development and project implementation.

### 3.4.1 Technical Feasibility:

The technology used can be developed with the current equipment's and has the technical capacity to hold the data required by the new system.

- This technology supports the modern trends of technology.
- Easily accessible, more secure technologies.

Technical feasibility on the existing system and to what extent it can support the proposed addition. We can add new modules easily without affecting the Core Program. Most of parts are running in the server using the concept of stored procedures.

### 3.4.2 Economic Feasibility

- Analysis of a project's costs and revenues in an effort to determine whether or not it is logical and possible to complete
- Minimum requirements in terms of the software specification due to the open source software tool MQTTX, Node.js packages which makes it economically feasible.
- In the case of memory overflow, we need to prove a hard disk as secondary memory which will not cost much.
- Hence the project is economically feasible.

### 3.4.3 Operational Feasibility

The aspect of study is to check the level of acceptance of the system by the user. This includes the process of training the user to use the system efficiently. The user must not feel threatened by the system, instead must accept it as a necessity. The level of acceptance by the users solely depends on the methods that are employed to educate the user about the system and to make him familiar with it. His level of confidence must be raised so that he is also able to make some constructive criticism, which is welcomed, to conclude that our project is operationally feasible we analysed the following issues.

1.Does the proposed system overcome existing drawbacks? (yes, in the existing system only one algorithm is used here ensemble is used which combines algorithms and produces more accuracy)

2.Does the proposed system pass the minimum acceptance criteria? (yes, the proposed system provides better results than the existing system).

As a result, we conclude that our project is operationally feasible.

## CHAPTER 4 SYSTEM DESIGN AND REQUIREMENTS

### 4.1 SYSTEM DESIGN

The most challenging phase of the system development process is the system design. It becomes the foundation for understanding the procedural details necessary for implementing the system recommended in the feasibility study. Design involves a decent amount of thought processing in terms of logical and physical stages of development. In designing a new system, the system analyst must have a clear understanding of the objectives which it is supposed to fulfil.

System design is the process of defining the architecture, interfaces, and data for a system that satisfies specific requirements. System design meets the needs of your business or organization through coherent and efficient systems.

System design includes activities to conceive a set of system elements that answers a specific, intended purpose, using principles and concepts; it includes assessments and decisions to select system elements that compose the system, fit the architecture of the system, and comply with traded-off system requirements.

The design phase mainly comprises of system architecture diagram, module-wise context diagrams followed by individual module description. The following system architecture diagram has been proposed for the project-  
Research on Secure Communication of Esp32 IoT Embedded System to Node JS Cloud server using HTTPS/MQTT.

#### 4.1.1 SYSTEM ARCHITECTURE

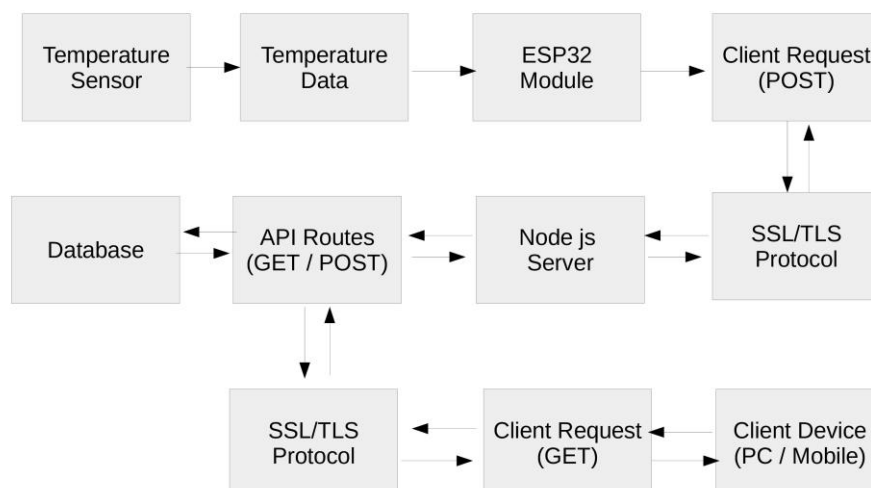


Fig 4.1.1 System Architecture

The figure 4.1 represents the system architecture of the proposed project- Research on Secure Communication of Esp32 IoT Embedded System to Node JS Cloud server using HTTPS/MQTT.

#### 4.1.2 DATA FLOW DIAGRAM

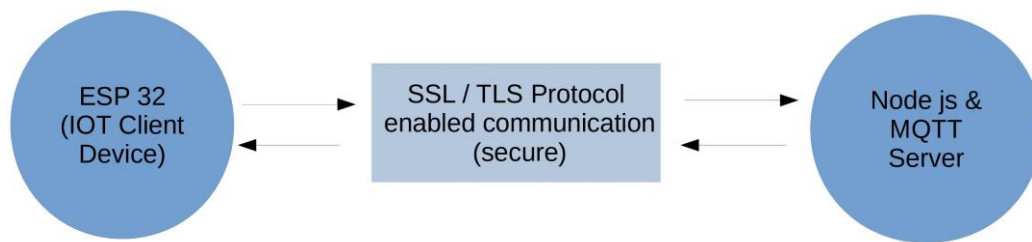


Fig 4.1.2 SSL/TLS Protocol enabled communication

#### 4.1.3 MODULE DESCRIPTION

- Node js / MQTT Server implementation module
- Route Setuping module
- Data posting and fetching module

##### 4.1.3.1 Node js Server Implementation

As we discussed earlier in system configurations the Node js is installed on the server system. The program setup is given below:

```

var app = require('express')();
var https = require('https');
var fs = require('fs');
const mysql = require('mysql');
const connection = mysql.createConnection({
  "host" : process.env.db_host,
  "user" : process.env.db_user,
  "password" : process.env.db_password,
  "database" : process.env.db_database,
  "multipleStatements": true
});

connection.connect(function(err) {
  if (err) {
    console.error('error connecting: ' + err.stack);
    return;
  }
  console.log('connected as id ' + connection.threadId);
});

var server = https.createServer({
  key: fs.readFileSync('privkey.pem'),
  cert: fs.readFileSync('fullchain.pem')
}, app);
server.listen(3025);

app.get("/", function(request, response){
  response.send("success");
});

app.get("/get-plant-data", function(request, response){
  connection.query("select * from plant_data", null, function(err, result){
    if(err) return response.status(500).json({ message: err });
    return response.status(200).json(data);
  });
});
  
```



```

});
});

app.post("/post-plant-data", function(request, response){
  const v = request.body;
  const insert_sql = "insert into inverter_data (rdate, plant_id, temperature) values (?, ?, ?)";
  connection.query(insert_sql, [v.rdate, v.plant_id, v.temperature], function(err, result){
    if(err) return response.status(500).json({ message: err });
    return response.status(201).json({ message: "Plant data uploaded successfully" });
  });
});
});

```

Program 4.3.1 Node js server implementation

#### 4.1.3.2 Route Setuping Module

##### (i) GET method

method – GET

route - /get-plant-data

This method triggers when the end user wants to get the temperature of the plant date and made a request call to above mentioned API from the client device (Mobile / PC / Laptop / Tablet) which is a web-browser based device. It invokes a database call and sends back a response.

If fetch request succeeds it gives back a response with success status code (201) and with the plant-data to the client.

If failed, then it response back with the error report with status code (500).

##### (ii) POST method

method – POST

route - /post-plant-data

request body = temperature data

This method is invoked when the Client (IoT device ESP32) tries to send the temperature data to the server through the above mentioned end point with data as request body.

When the temperature data is fetched from the request body the above program makes a database call to insert the given data in the target database.

If succeed it gives back a response with success status code (201) and with a simple message;

If failed, then it response back with the error report with status code (500).

#### 4.1.3.3 Data posting and fetching module

##### (i) Fetch:

The Fetch API provides an interface for fetching resources (including across the network). It will seem familiar to anyone who has used XMLHttpRequest, but the new API provides a more powerful and flexible feature set.

Fetch provides a generic definition of Request and Response objects (and other things involved with network requests). This will allow them to be used wherever they are needed in the future, whether it's for service workers, Cache API, and other similar things that handle or modify requests and responses, or any kind of use case that might require you to generate your responses programmatically (that is, the use of computer program or personal programming instructions).

It also defines related concepts such as CORS and the HTTP Origin header semantics, supplanting their separate definitions elsewhere.

In our case, the plant data can be fetched by making a **fetch** request to the API route definition with its URL.

##### fetch()

The fetch() method used to fetch a resource.

Eg: fetch(<https://www.greatindianmovies/slumdog-millionaire>)

##### Headers

Represents response/request headers, allowing you to query them and take different actions depending on the results.

##### Request

Represents a resource request.

##### Response

Represents the response to a request.

##### (ii) POST:

The HTTP POST method sends data to the server. The type of the body of the request is indicated by the Content-Type header.

The difference between PUT and POST is that PUT is idempotent: calling it once or several times successively has the same effect (that is no side effect), where successive identical POST may have additional effects, like passing an order several times.

A POST request is typically sent via an HTML form and results in a change on the server. In this case, the content type is selected by putting the adequate string in the enctype attribute of the <form> element or the formenctype attribute of the <input> or <button> elements:

**application/x-www-form-urlencoded:** The keys and values are encoded in key-value tuples separated by '&', with a '=' between the key and the value. Non-alphanumeric characters in both keys and values are percent encoded: this is the reason why this type is not suitable to use with binary data (use multipart/form-data instead)

**multipart/form-data:** Each value is sent as a block of data ("body part"), with a user agent-defined delimiter ("boundary") separating each part. The keys are given in the Content-Disposition header of each part.

**text/plain:** When the POST request is sent via a method other than an HTML form like via an XMLHttpRequest — the body can take any type. As described in the HTTP 1.1 specification, POST is designed to allow a uniform method to cover the following functions:

Annotation of existing resources

Posting a message to a bulletin board, newsgroup, mailing list, or similar group of articles;

Adding a new user through a signup modal;

Providing a block of data, such as the result of submitting a form, to a data-handling process;

Extending a database through an append operation.

In our case, the post request is made with the temperature data as a request body to the above mentioned API endpoint.

## 4.2 SYSTEM REQUIREMENTS

This chapter discusses about the hardware and software requirements and also the configuration management of the system under sections 4.1, 4.2 and 4.3 respectively.

### 4.2.1 HARDWARE SPECIFICATIONS

- ESP-32 Development kit,
- Cloud / Remote server with minimum of 2GB RAM,
- Windows / Linux / mac OS.

### 4.2.2 SOFTWARE REQUIREMENT

- Node js v16.x.x,
- Mosquitto Server,
- MQTTX Mosquitto Client,
- SQL / NoSQL Database,
- JSON Library to parse data,
- Encryption techniques.

### 4.2.3 CONFIGURATIONS

Software Configuration means any of the Deliverables set forth in the Initial Order Form related to the configuration or adaptation of the Software or the Standard Reports or Templates within the Software or the creation of business rules using the Software.

The system needs to be configured under both hardware and software to work accordingly. In our system there are two sides needed to be configured. They are,

- (i) Client side configuration and
- (ii) Server side configuration.

#### 4.2.3.1 Client Side Configuration

##### 4.2.3.1.1 ESP32 Dev kit

ESP32-S2-DevKitM-1 is entry-level development board. Most of the I/O pins on the module are broken out to the pin headers on both sides for easy interfacing. Developers can either connect peripherals with jumper wires or mount ESP32-S2-DevKitM-1 on a breadboard.



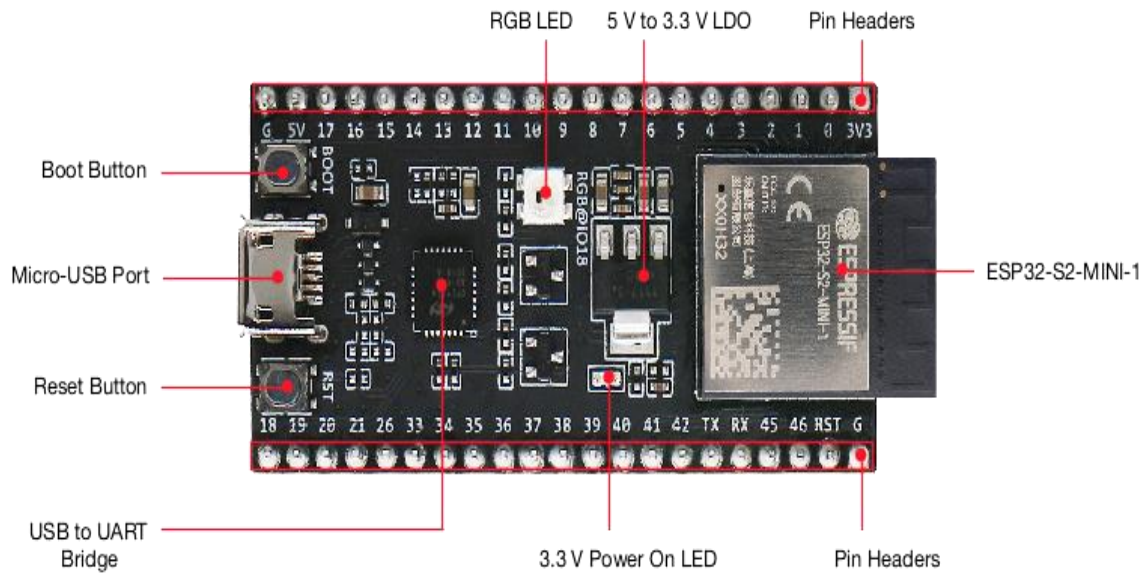


Fig. 4.2.3.1.1 ESP32 Mini S1

Key component	Description
On-board module (ESP32-S2-MINI-1 or ESP32-S2-MINI-1U in figures above)	ESP32-S2-MINI series modules with an on-board PCB antenna or a connector for an external antenna. This series of modules, known for its small size, have a flash and/or a PSRAM integrated in the chip package. For more information, please refer to Ordering Information.
Key component	Description
Pin Headers	All available GPIO pins (except for the SPI bus for flash) are broken out to the pin headers on the board. Users can program ESP32-S2FH4 chip to enable multiple functions such as SPI, I2S, UART, I2C, touch sensors, PWM etc. For details, please see Header Block.
3.3 V Power On LED	Turns on when the USB power is connected to the board.
USB to UART Bridge	Single USB-UART bridge chip provides transfer rates up to 3 Mbps.
Reset Button	Reset button.
Micro-USB Port	USB interface. Power supply for the board as well as the communication interface between a computer and the ESP32-S2FH4 chip.
Boot Button	Download button. Holding down <b>Boot</b> and then pressing <b>Reset</b> initiates Firmware Download mode for downloading firmware through the serial port.
RGB LED	Addressable RGB LED, driven by GPIO18.
5 V to 3.3 V LDO	Power regulator that converts a 5 V supply into a 3.3 V output.
External Antenna Connector	On <b>ESP32-S2-MINI-2U</b> and <b>ESP32-S2-MINI-1U</b> module only. For connector dimensions, please refer to Section External Antenna Connector Dimensions in module datasheet.
Table 4.2.3.1.2 ESP32 Mini S1	

### Working

ESP32 can perform as a complete standalone system or as a slave device to a host MCU, reducing communication stack overhead on the main application processor. ESP32 can interface with other systems to provide Wi-Fi and Bluetooth functionality through its SPI / SDIO or I2C / UART interfaces.

Here our ESP32 module can send the temperature data read from the temperature sensor to our server periodically.

#### 4.2.3.1.2 ESP-32

##### 4.2.3.1.2.1 Features

ESP32 has a unique MAC address assigned by the manufacturer, which is burned into the device's non-volatile memory during the manufacturing process. The MAC address is a globally unique identifier that is used to identify the device on the network.

The MAC address of an ESP32 is not typically used in encryption. The MAC address is primarily used for identifying the device on the network and is not directly related to the encryption process.

Encryption typically involves using cryptographic algorithms and keys to protect sensitive data and prevent unauthorized access. The specific algorithms and keys used will depend on the encryption method being used, such as AES, RSA, or SHA. The ESP32 can use various cryptographic libraries, such as embed TLS, to implement encryption and decryption.

##### 4.2.3.1.2.2 Variants

There are many ESP32 modules available from various manufacturers, and most of them have similar specifications in terms of processing power, memory, and networking capabilities. When it comes to SSL/TLS performance, there may be some variation based on the specific implementation and hardware acceleration available on the module.

That being said, some popular ESP32 modules with good SSL/TLS performance include:

1. ESP32-WROVER-B: This module has 4 MB of external PSRAM and supports hardware acceleration for AES, SHA-2, and RSA encryption/decryption. It also has a built-in 802.11 b/g/n Wi-Fi transceiver.
2. ESP32-WROOM-32D: This module has 4 MB of embedded flash memory and supports hardware acceleration for AES, SHA-2, and RSA encryption/decryption. It also has a built-in 802.11 b/g/n Wi-Fi transceiver.
3. ESP32-PICO-D4: This module has 4 MB of embedded flash memory and supports hardware acceleration for AES, SHA-2, and RSA encryption/decryption. It also has a built-in 802.11 b/g/n Wi-Fi transceiver and Bluetooth 4.2.

##### 4.2.3.1.2.3 Encryption

The encrypted content will not be the same across devices, even if the same encryption algorithm and key are used. This is because the encryption process is designed to produce different output (ciphertext) for the same input (plaintext) each time it is run. This is done by adding a random initialization vector (IV) to the input before encryption, which ensures that even identical plaintext inputs will produce different ciphertext outputs.

Furthermore, even if the IV and key were somehow the same across devices, the ciphertext would still be different due to potential differences in the implementation of the encryption algorithm, differences in the input data, and other factors. Therefore, it is not possible to rely on identical ciphertext outputs across devices as a means of achieving any sort of consistency or interoperability.

##### 4.2.3.1.2.4 Decryption

To decrypt the ciphertext generated by an encryption algorithm, we will need the same key and initialization vector (IV) that were used during the encryption process. If we have these, we can pass the ciphertext along with the key and IV to the decryption algorithm to obtain the original plaintext.

It is important to note that the IV used during encryption must be the same IV used during decryption, and that the key must be kept secure to prevent unauthorized access to the plaintext. Additionally, some encryption algorithms and modes require additional information, such as a nonce or authentication tag, to be passed along with the ciphertext and key in order to successfully decrypt the data.

In general, it is important to follow established best practices for key management and data encryption in order to ensure the security and integrity of sensitive data.

If we're using HTTPS with an ExpressJS server, we don't need to pass any encryption keys or initialization vectors to the server at design time. The keys and IVs are used in the encryption process on the client side (in this case, the ESP32 device) and the decryption process on the server side (in this case, the ExpressJS server).

When we set up HTTPS with an ExpressJS server, we typically provide the server with a certificate and a private key, which are used to authenticate and encrypt the connection between the client and the server. These keys are typically generated separately, and then securely stored on the server. The client (in this case, the ESP32) will need to have access to the public key in order to establish a secure connection to the server.

So, in summary, we do not need to pass any keys or IVs to the server at design time when setting up HTTPS with ExpressJS. we will need to ensure that the server has a valid certificate and private key, and that the client (ESP32) has access to the server's public key in order to establish a secure connection.

#### 4.2.3.1.2.5 Key and Initialization vector

The key and initialization vector (IV) are important components in many encryption algorithms.

The key is a secret value that is used to encrypt and decrypt data. It must be kept confidential, as anyone with knowledge of the key can read or modify the encrypted data.

The IV is a random value that is used to ensure that two identical plaintext blocks encrypt to different ciphertext blocks. It is typically a random value that is generated for each encryption operation.

In some encryption modes, the IV is combined with the key to produce a unique "key-IV" pair that is used to encrypt and decrypt data. In other encryption modes, the IV is used independently of the key.

The specific key and IV values used in the encryption process depend on the encryption algorithm and mode being used. It is important to use strong and random keys and IVs to ensure the security of the encrypted data.

#### 4.2.3.1.2.6 WiFiClientSecure and IV

When using the WiFiClientSecure library, the key and initialization vector (IV) are automatically generated and used for encryption and decryption. The library handles the entire encryption process, including generating and using the necessary keys and IVs, so we do not have to worry about it.

#### 4.2.3.1.2.7 RISC-V ISA

RISC-V is an open-source instruction set architecture (ISA) that is gaining popularity in the microcontroller and embedded systems community. The ESP32 is a microcontroller made by Espressif Systems, which has a dual-core processor and supports Wi-Fi and Bluetooth connectivity.

The RISC-V ESP32 is an ESP32 microcontroller that has been modified to run the RISC-V instruction set architecture. This allows developers to write software for the ESP32 using the RISC-V ISA instead of the proprietary Xtensa ISA used by the original ESP32.

The RISC-V ESP32 is still a relatively new development, and its availability and stability may vary depending on the development board or platform being used. However, the RISC-V ISA is gaining popularity in the industry and academia as an open, extensible, and scalable alternative to proprietary ISAs.

#### What is ISA

The Instruction Set Architecture (ISA) of a processor is not directly related to SSL/TLS (Secure Sockets Layer/Transport Layer Security), which are cryptographic protocols used for secure communication over the internet.

However, the performance of SSL/TLS operations on a processor can be affected by its ISA. For example, processors with hardware acceleration for encryption and decryption operations can perform SSL/TLS operations faster than processors without such hardware.

Additionally, the availability of cryptographic libraries and support for SSL/TLS protocols on a particular processor may depend on its ISA. For example, the availability and quality of SSL/TLS libraries for the RISC-V ISA may differ from those available for the Xtensa ISA.

In general, the choice of processor and ISA may be one of many factors to consider when implementing SSL/TLS on an embedded system. Other factors may include the specific use case, available memory and processing power, security requirements, and compatibility with other hardware and software components.

#### RISC-V ISA vs Xtensa ISA

The RISC-V ISA and the Xtensa ISA are both instruction set architectures designed for embedded systems, such as the ESP32. While both ISAs have their own advantages and disadvantages, here are some potential benefits of the RISC-V ISA over the Xtensa ISA:

- **Open-source architecture:** The RISC-V ISA is open-source, which means that anyone can access and modify the architecture. This can lead to a more collaborative development environment, where engineers from different companies and backgrounds can contribute to the development of the architecture. In contrast, the Xtensa ISA is proprietary and owned by Cadence Design Systems, which may limit its accessibility and modifiability.

- **Customizability:** The RISC-V ISA is designed to be highly customizable, which means that it can be tailored to specific use cases or optimized for particular applications. This can potentially lead to better performance or energy efficiency for embedded systems. In contrast, the Xtensa ISA is also highly customizable but may require a license from Cadence Design Systems to access certain features.

- **Standardization:** The RISC-V ISA is gaining popularity and is becoming more standardized in the industry, with a growing number of companies and organizations adopting it. This can lead to a more consistent development environment and a larger pool of available tools and libraries. In contrast, the Xtensa ISA is less widely adopted and may have a smaller pool of available tools and libraries.

It is important to note that the specific benefits of the RISC-V ISA over the Xtensa ISA may depend on the specific use case and implementation. Additionally, the ESP32 RISC-V processor is still relatively new and its performance and stability may not be as well established as the Xtensa processor.

#### • RISC-V for HTTPS

Whether RISC-V is better for HTTPS than other architectures depends on various factors such as the performance of the processor, the amount of memory available, and the availability of optimized software libraries.

In general, RISC-V processors can be optimized for specific tasks by designing custom instructions, which can potentially improve performance for specific applications such as HTTPS. However, the actual performance gains will depend on the specific implementation and optimization of the processor.

That being said, the ESP32 RISC-V processor is still a relatively new development, and its performance and stability may not be as well established as other architectures. Additionally, the availability of software libraries and development tools for the RISC-V architecture may be more limited than for other architectures.

Ultimately, the choice of architecture for HTTPS will depend on various factors such as the specific requirements of the application, the available hardware and software resources, and the expertise and familiarity of the development team with different architectures.

#### • WiFiClientSecure with RISC-V

The WiFiClientSecure library can be used on an ESP32 based on the RISC-V ISA, as long as the necessary libraries and dependencies are installed and configured properly.

The RISC-V architecture does not affect the compatibility or functionality of the WiFiClientSecure library, which is a software library that implements secure communication protocols such as TLS/SSL for the ESP32. The library relies on the hardware and firmware features of the ESP32, rather than the specific architecture or instruction set used by the processor.

However, it is important to note that the performance and stability of the WiFiClientSecure library on a RISC-V based ESP32 may depend on the specific implementation and optimization of the RISC-V processor, as well as the availability and compatibility of other software libraries and dependencies required for the ESP32.

#### 4.2.3.1.2.7 Using RTC

In the context of SSL/TLS, an RTC can be useful to ensure that the client and server have synchronized clocks, which is important for the proper functioning of SSL/TLS certificates. SSL/TLS certificates have a start and end date, and the certificate validation process involves comparing the current date and time to the start and end dates of the certificate. If the clocks of the client and server are not synchronized, certificate validation may fail and the SSL/TLS connection may not be established.

However, it is worth noting that many devices, including the ESP32, have an internal clock that can keep reasonably accurate time, so an external RTC may not always be necessary. If our ESP32 device has a reliable source of time, such as an NTP server or GPS module, we may not need an RTC.

#### 4.2.3.1.2..8 WiFiClientSecure and RTC

WiFiClientSecure library will not automatically check the date and time from the RTC (Real Time Clock) module. It uses the date and time provided by the system clock of the device, which can be inaccurate if the clock is not synchronized with a reliable time source like NTP (Network Time Protocol).

To ensure that the correct date and time are used for SSL/TLS certificate validation, we need to manually synchronize the RTC with an NTP server at regular intervals (e.g., once a day). we can use the NTPClient library in Arduino to synchronize the RTC with an NTP server. Once the RTC is synchronized, we can retrieve the current date and time from the RTC and use it to validate SSL/TLS certificates.

#### 4.2.3.1.2.9 NTP instead of RTC

NTP (Network Time Protocol) is a protocol used to synchronize the time on network-connected devices. we can use NTP to synchronize the time on our ESP32 instead of an RTC (Real-Time Clock) module. Here are the steps to use NTP on ESP32:

- Include the NTPClient library in our Arduino IDE.

```
#include <NTPClient.h>
```

- Create a WiFiUDP object to manage the UDP connection required for NTP.

```
WiFiUDP ntpUDP;
```

- Create an NTPClient object, passing in the ntpUDP object and the NTP server address.

```
NTPClient timeClient(ntpUDP, "pool.ntp.org");
```

- In the setup() function, initialize the WiFi connection and set the time zone of our location.

```
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(1000);
}
timeClient.begin();
timeClient.setTimeOffset(19800); // GMT+5:30
```

In the loop() function, update the time from the NTP server at regular intervals and use it as required in our code.

```
timeClient.update();
Serial.println(timeClient.getFormattedTime());
```

With these steps, our ESP32 will synchronize the time with the NTP server and keep the time up-to-date. we can then use this time to provide time-stamping in our SSL/TLS connection or any other time-sensitive application.

#### 4.2.3.1.2.10 RTC sync with NTP

To synchronize the real-time clock (RTC) with an NTP server, we need to perform the following steps:

- Connect to an NTP server and retrieve the current time.
- Set the RTC with the current time received from the NTP server.
- Periodically update the RTC by adjusting it with the time received from the NTP server.

Here's a code example that demonstrates how to synchronize the RTC with an NTP server using the ESP32 and the NTPClient library:

```
#include <WiFi.h>
#include <NTPClient.h>
#include <WiFiUdp.h>
#include <RTClib.h>

const char* ssid = "our_SSID";
const char* password = "our_PASSWORD";

const long utcOffsetInSeconds = 0;
WiFiUDP ntpUDP;
NTPClient timeClient(ntpUDP, "pool.ntp.org", utcOffsetInSeconds);

RTC_DS3231 rtc; // change to RTC_PCF8523 for PCF8523

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
```



```

}

// Set RTC time with NTP time
timeClient.begin();
while(!timeClient.update()) {
  timeClient.forceUpdate();
}
rtc.begin();
rtc.adjust(DateTime(timeClient.getEpochTime()));

// Other setup code here
}

void loop() {
  // Update RTC time every 24 hours
  if (millis() % (24 * 60 * 60 * 1000) == 0) {
    timeClient.update();
    rtc.adjust(DateTime(timeClient.getEpochTime()));
  }

  // Other loop code here
}

```

In this example, we first connect to the WiFi network and initialize an NTPClient object to retrieve the current time from the NTP server. We then use the update() method of the NTPClient object to retrieve the time from the NTP server and adjust the RTC using the adjust() method of the RTCLib library. Finally, we periodically update the RTC time every 24 hours by comparing the current millisecond count with a multiple of 24 hours.

#### 4.2.3.1.2.11 NTP / RTC update frequency

If our application only needs to check the time once a day, then fetching the time from an NTP server once during setup should be sufficient. We can use the time() function in the time.h library to get the current time in seconds since the Unix epoch (January 1, 1970, 00:00:00 UTC).

We can then store this value in a variable and use it throughout our application to compare against the expiration dates of our certificates. Keep in mind that if the ESP32 loses power or is reset, the time value will be lost and we'll need to fetch the time from the NTP server again during setup.

Alternatively, we can use a real-time clock (RTC) module to keep track of time even when the ESP32 loses power or is reset. We can then set up the RTC to sync with an NTP server once a day, and use the time value from the RTC in our application. This can reduce the frequency of requests to the NTP server and provide more accurate timekeeping.

#### 4.2.3.1.2.12 RTC supported Variants

1. The ESP32-WROVER module includes an internal RTC (Real-Time Clock) with a separate power supply domain to maintain accurate timekeeping even when the rest of the system is in deep sleep mode. So, yes, the WROVER module has an internal RTC.

2. The WROOM-32D is a module based on the ESP32 microcontroller and does not have an internal RTC. However, it has an external interface for connecting an external RTC chip if needed.

3. The ESP32-PICO-D4 is a System-in-Package (SiP) module that integrates an ESP32 chip, 4 MB SPI flash, a crystal oscillator, filter capacitors, and other components needed to implement a complete WiFi and Bluetooth connectivity solution. The ESP32-PICO-D4 does not include an internal RTC, but it can be connected to an external RTC module or use an NTP (Network Time Protocol) server to synchronize its clock.

#### 4.2.3.1.2.13 Client Side Points

Here are some additional points to consider when implementing SSL/TLS on the client side:

- **Verify the server's certificate:**

When establishing an SSL/TLS connection, the client should verify the server's certificate to ensure that it is valid and issued by a trusted certificate authority (CA). This can help prevent man-in-the-middle attacks and other security threats. The WiFiClientSecure library in ESP32 includes built-in certificate verification functionality.

- **Use strong cipher suites:**



The cipher suite is a combination of encryption algorithms, key exchange algorithms, and message authentication codes used to secure the communication channel. It is important to use strong cipher suites to ensure the security of the connection. The WiFiClientSecure library supports a variety of cipher suites.

- **Use secure protocols:**

The SSL and TLS protocols have evolved over time to address known security vulnerabilities. It is important to use the most recent and secure version of the protocol supported by both the client and server.

- **Protect private keys:**

If the client is using client-side certificates for authentication, it is important to protect the private key associated with the certificate. Private keys should be stored in a secure location and protected with a strong passphrase.

- **Use secure storage:**

If the client needs to store sensitive information, such as private keys or passwords, it should be stored in a secure location. ESP32 includes secure storage options, such as the SPIFFS file system and the EEPROM library.

- **Perform regular updates:**

It is important to keep the client software up to date with the latest security patches and updates to ensure that any known vulnerabilities are addressed.

#### 4.2.3.2 Server Side Configuration

Server side configuration refers to operations that are performed by the server in a client server relationship in a computer network.

We are using two types of servers namely HTTPS server and MQTT server.

##### 4.2.3.2.1 HTTPS Server Configuration

###### 4.2.3.2.1.1 Node js Libraries

The library in the development of NodeJS applications is the best perfect for developers who use libraries, Node JS Packages, and scripts.

###### 4.2.3.2.1.1.1 Express

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

With a myriad of HTTP utility methods and middleware at your disposal, creating a robust API is quick and easy. Express provides a thin layer of fundamental web application features, without obscuring Node.js features that you know and love.

- This is a Node.js module available through the npm registry.
- Installation is done using the npm install command:
  - \$ npm install express

#### Express Configurations:

##### API Routes:

###### (i) Method – GET - “/get-plant-data”

This GET method is used to get the data from the server in case if we want to know that our uploaded datas are successfully storing in the database. And also for data validation purposes.

###### (ii) Method – POST- “/post-plant-data”

The POST method is used for sending the monitored temperature data from our client side (IoT device) to server.

###### 4.2.3.1.1.2 HTTPS

The leanest and most handsome HTTP client in the Node.js.

- Installation is done using the npm install command:
  - \$ npm install https

###### 4.2.3.2.1.2 Self-signed certificates

A self-signed certificate is one that is not signed by a CA at all – neither private nor public. In this case, the certificate is signed with its own private key, instead of requesting it from a public or a private CA.

#### Certbot

Certbot is usually meant to be used to switch an existing HTTP site to work in HTTPS (and, afterward, to continue renewing the site's HTTPS certificates whenever necessary). Some Certbot documentation assumes or recommends that you have a working web site that can already be accessed using HTTP on port 80.

**Let's Encrypt** is a free, automated, and open certificate authority brought to you by the nonprofit Internet Security Research Group (ISRG).

Let's Encrypt is a Certificate Authority (CA) that provides a way to obtain and install free TLS/SSL certificates, thereby enabling encrypted HTTPS on web servers. It streamlines the process by providing a software client, Certbot, that attempts to automate most (if not all) of the required steps.

#### Certbot installation:

- `$ sudo apt install certbot python3-certbot-nginx`
- `$ sudo certbot --nginx -d example.com -d --nginx -d www.example.com`
- `$ server_name example.com www.example.com`
- `$ sudo nginx -t`
- `$ sudo systemctl reload nginx`
- `$ sudo ufw status`
- `$ sudo ufw allow 'Nginx Full'`
- `$ sudo ufw delete allow 'Nginx HTTP'`
- `$ sudo ufw status`

#### Example Output:

Status: active

To	Action	From
--	-----	----
OpenSSH	ALLOW	Anywhere
Nginx Full	ALLOW	Anywhere
OpenSSH (v6)	ALLOW	Anywhere (v6)
Nginx Full (v6)	ALLOW	Anywhere (v6)

- `$ sudo certbot --nginx -d example.com -d www.example.com`

#### Example Output:

##### IMPORTANT NOTES:

- Congratulations! Your certificate and chain have been saved at:  
/etc/letsencrypt/live/example.com/fullchain.pem  
Your key file has been saved at:  
/etc/letsencrypt/live/example.com/privkey.pem  
Your cert will expire on 2023-08-18. To obtain a new or tweaked version of this certificate in the future, simply run certbot again with the "certonly" option. To non-interactively renew \*all\* of your certificates, run "certbot renew"
- If you like Certbot, please consider supporting our work by:

Donating to ISRG / Let's Encrypt: <https://letsencrypt.org/donate>  
Donating to EFF: <https://eff.org/donate-le>

We installed the Let's Encrypt client certbot, downloaded SSL certificates for your domain, configured Nginx to use these certificates.

#### 4.2.3.2.2 MQTT Server Configuration

Message Queue Telemetry Transport (MQTT) is an OASIS standard messaging protocol for the Internet of Things (IoT). It is designed as an extremely lightweight publish/subscribe messaging transport that is ideal for connecting remote devices with a small code footprint and minimal network bandwidth.

MQTT is a machine-to-machine messaging protocol, designed to provide lightweight publish/subscribe communication to "Internet of Things" devices.

#### Mosquitto Client installation on MQTT Server

##### Step 1 — Installing Mosquitto

- `$ sudo apt-get install mosquitto mosquitto-clients`
- `$ mosquitto_sub -h localhost -t test`
- `$ mosquitto_pub -h localhost -t test -m "hello world"`

**Step 2 — Installing Certbot for Let's Encrypt Certificates**

- `$ sudo add-apt-repository ppa:certbot/certbot`
- `$ sudo apt-get update`
- `$ sudo apt-get install certbot`

**Step 3 — Running Certbot**

- `$ sudo ufw allow http`
- `$ sudo certbot certonly --standalone --standalone-supported-challenges http-01 -d mqtt.example.com`

**Step 4 — Setting up Certbot Automatic Renewals**

- `$ sudo crontab -e`
- `$ crontab`

**Step 5 — Configuring MQTT Passwords**

- `$ sudo mosquitto_passwd -c /etc/mosquitto/passwd sammy`
- `$ sudo nano /etc/mosquitto/conf.d/default.conf`
- `$ sudo systemctl restart mosquitto`
- `$ mosquitto_pub -h localhost -t "test" -m "hello world"`
- `$ mosquitto_sub -h localhost -t test -u "sammy" -P "password"`
- `$ mosquitto_pub -h localhost -t "test" -m "hello world" -u "sammy" -P "password"`

**Step 6 — Configuring MQTT SSL**

- `$ sudo nano /etc/mosquitto/conf.d/default.conf`

```
/etc/mosquitto/conf.d/default.conf
```

```
...
```

```
listener 1883 localhost
```

```
listener 8883
```

```
certfile /etc/letsencrypt/live/mqtt.example.com/cert.pem
```

```
cafile /etc/letsencrypt/live/mqtt.example.com/chain.pem
```

```
keyfile /etc/letsencrypt/live/mqtt.example.com/privkey.pem
```

Save and exit the file, then restart Mosquitto to update the settings:

- `$ sudo systemctl restart mosquitto`

Update the firewall to allow connections to port 8883.

- `$ sudo ufw allow 8883`

**Step 7 — Configuring MQTT Over Websockets (Optional)**

- `$ sudo nano /etc/mosquitto/conf.d/default.conf`
- `$ sudo systemctl restart mosquitto`
- `$ sudo ufw allow 8083`

To securely post data to an MQTT server using the ESP32, we can use the PubSubClient library along with the WiFiClientSecure library. Here's an example code snippet:

```
#include <WiFi.h>
```

```
#include <WiFiClientSecure.h>
```

```
#include <PubSubClient.h>
```

```
const char* ssid = "our_SSID";
```

```
const char* password = "our_PASSWORD";
```

```
const char* mqtt_server = "our_MQTT_server";
```

```
const int mqtt_port = 8883; // or the port of our MQTT server
```

```
const char* mqtt_user = "our_MQTT_username";
```

```
const char* mqtt_password = "our_MQTT_password";
```

```
const char* root_ca_cert = "our_root_CA_certificate";
```

```
WiFiClientSecure espClient;
```

```
PubSubClient client(espClient);
```

```
void setup() {
```

```

Serial.begin(115200);
WiFi.begin(ssid, password);
while (WiFi.status() != WL_CONNECTED) {
  delay(1000);
  Serial.println("Connecting to WiFi...");
}
Serial.println("Connected to WiFi");

// Set the root CA certificate
espClient.setCACert(root_ca_cert);

// Set up the MQTT client
client.setServer(mqtt_server, mqtt_port);
client.setCallback(callback);
}

void loop() {
  // Connect to the MQTT server
  if (!client.connected()) {
    Serial.println("Connecting to MQTT server...");
    if (client.connect("ESP32Client", mqtt_user, mqtt_password)) {
      Serial.println("Connected to MQTT server");
      client.subscribe("test/topic");
    } else {
      Serial.print("Connection failed: ");
      Serial.println(client.state());
      delay(5000);
      return;
    }
  }

  // Publish a message
  client.publish("test/topic", "Hello, world!");

  // Wait for messages to be received
  client.loop();
}

void callback(char* topic, byte* payload, unsigned int length) {
  // Handle incoming messages here
}

```

In the example code, a `WiFiClientSecure` object is created and the root CA certificate is set using the `setCACert` function. The `PubSubClient` object is created and the MQTT server is set using the `setServer` function. The `connect` function is used to establish a secure connection to the MQTT server, and the `publish` function is used to send a message to the server. The `loop` function is called to handle incoming messages.

Note that we will need to replace the placeholders for `ssid`, `password`, `mqtt_server`, `mqtt_user`, `mqtt_password`, and `root_ca_cert` with our own values.

Also note that the MQTT server must be configured to support secure connections over TLS/SSL, and we will need to obtain the root CA certificate for our MQTT server or certificate authority.

#### 4.2.4 SOFTWARE DESCRIPTION

##### 4.2.4.1 Node.js v18.12.1 LTS



Fig 4.2.4.1 Node js Logo

Node.js is an open-source server environment. Node.js is cross-platform and runs on Windows, Linux, Unix, and macOS. Node.js is a back-end JavaScript runtime environment. Node.js runs on the V8 JavaScript Engine and executes JavaScript code outside a web browser.

Node.js lets developers use JavaScript to write command line tools and for server-side scripting. The functionality of running scripts server-side produces dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js represents a "JavaScript everywhere" paradigm, unifying web-application development around a single programming language, rather than different languages for server-side and client-side scripts.

Node.js has an event-driven architecture capable of asynchronous I/O. These design choices aim to optimize throughput and scalability in web applications with many input/output operations, as well as for real-time Web applications (e.g., real-time communication programs and browser games).

The Node.js distributed development project was previously governed by the Node.js Foundation, and has now merged with the JS Foundation to form the OpenJS Foundation. OpenJS Foundation is facilitated by the Linux Foundation's Collaborative Projects program.

- There are thousands of open-source libraries for Node.js, most of them hosted on the npm website. There are multiple developer conferences and events that support the Node.js community, including NodeConf, Node Interactive, and Node Summit as well as a number of regional events.
- The open-source community has developed web frameworks to accelerate the development of applications. Such frameworks include Connect, Express.js, Socket.IO, Feathers.js, Koa.js, Hapi.js, Sails.js, Meteor, Derby, and many others. Various packages have also been created for interfacing with other languages or runtime environments such as Microsoft .NET

## V8

Main article: V8 (JavaScript engine)

V8 is the JavaScript execution engine which was initially built for Google Chrome. It was then open-sourced by Google in 2008. Written in C++, V8 compiles JavaScript source code to native machine code at runtime.[7] As of 2016, it also includes Ignition, a bytecode interpreter.

Package management

**npm** is the pre-installed package manager for the Node.js server platform. It installs Node.js programs from the npm registry, organizing the installation and management of third-party Node.js programs. Packages in the npm registry can range from simple helper libraries such as Lodash to task runners such as Grunt.

## Unified API

Node.js can be combined with a browser with your site, a database that supports JSON data (such as Postgres,[75] MongoDB, or CouchDB) and JSON for a unified JavaScript development stack. With the adaptation of what were essentially server-side development patterns such as MVC, MVP, MVVM, etc., Node.js allows the reuse of the same model and service interface between client side and server side.

## Event loop

Node.js registers with the operating system so the OS notifies it of connections and issues a callback. Within the Node.js runtime, each connection is a small heap allocation. Traditionally, relatively heavyweight OS processes or threads handled each connection. Node.js uses an event loop for scalability, instead of processes or threads.[76] In contrast to other event-driven servers, Node.js's event loop does not need to be called explicitly. Instead, callbacks are defined, and the server automatically enters the event loop at the end of the callback definition. Node.js exits the event loop when there are no further callbacks to be performed.

## WebAssembly

Node.js supports WebAssembly and as of Node 14 has experimental support of WASI, the WebAssembly System Interface.

### 4.2.4.2 HTTPS Server

Hypertext Transfer Protocol Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP). It is used for secure communication over a computer network, and is widely used on the Internet. In HTTPS, the communication protocol is encrypted using Transport Layer Security (TLS) or, formerly, Secure Sockets Layer (SSL). The protocol is therefore also referred to as HTTP over TLS, or HTTP over SSL.

The principal motivations for HTTPS are authentication of the accessed website, and protection of the privacy and integrity of the exchanged data while in transit. It protects against man-in-the-middle attacks, and the bidirectional encryption of communications between a client and server protects the communications against eavesdropping and tampering. The authentication aspect of HTTPS requires a trusted third party to sign server-side digital certificates. This was historically an expensive operation, which meant fully authenticated HTTPS connections were usually found only on secured payment transaction services and other secured corporate information systems on the World Wide Web. In 2016, a campaign by the Electronic Frontier Foundation with the support of web browser developers led to the protocol becoming more prevalent. HTTPS is now used more often by web users than the original non-secure HTTP, primarily to protect page authenticity on all types of websites; secure accounts; and to keep user communications, identity, and web browsing private.

## Security

*Main article: Transport Layer Security & Security:*

The security of HTTPS is that of the underlying TLS, which typically uses long-term public and private keys to generate a short-term session key, which is then used to encrypt the data flow between the client and the server. X.509 certificates are used to authenticate the server (and sometimes the client as well). As a consequence, certificate authorities and public key certificates are necessary to verify the relation between the certificate and its owner, as well as to generate, sign, and administer the validity of certificates. While this can be more beneficial than verifying the identities via a web of trust, the 2013 mass surveillance disclosures drew attention to certificate authorities as a potential weak point allowing man-in-the-middle attacks. An important property in this context is forward secrecy, which ensures that encrypted communications recorded in the past cannot be retrieved and decrypted should long-term secret keys or passwords be compromised in the future. Not all web servers provide forward secrecy.[needs update]

For HTTPS to be effective, a site must be completely hosted over HTTPS. If some of the site's contents are loaded over HTTP (scripts or images, for example), or if only a certain page that contains sensitive information, such as a log-in page, is loaded over HTTPS while the rest of the site is loaded over plain HTTP, the user will be vulnerable to attacks and surveillance. Additionally, cookies on a site served through HTTPS must have the secure attribute enabled. On a site that has sensitive information on it, the user and the session will get exposed every time that site is accessed with HTTP instead of HTTPS

## Difference from HTTP

HTTPS URLs begin with "https://" and use port 443 by default, whereas, HTTP URLs begin with "http://" and use port 80 by default.

HTTP is not encrypted and thus is vulnerable to man-in-the-middle and eavesdropping attacks, which can let attackers gain access to website accounts and sensitive information, and modify webpages to inject malware or advertisements. HTTPS is designed to withstand such attacks and is considered secure against them (with the exception of HTTPS implementations that use deprecated versions of SSL).

## Network layers

HTTP operates at the highest layer of the TCP/IP model—the application layer; as does the TLS security protocol (operating as a lower sublayer of the same layer), which encrypts an HTTP message prior to transmission and decrypts a message upon arrival. Strictly speaking, HTTPS is not a separate protocol, but refers to the use of ordinary HTTP over an encrypted SSL/TLS connection.

HTTPS encrypts all message contents, including the HTTP headers and the request/response data. With the exception of the possible CCA cryptographic attack described in the limitations section below, an attacker should at most be able to discover that a connection is taking place between two parties, along with their domain names and IP addresses.

## Server setup

To prepare a web server to accept HTTPS connections, the administrator must create a public key certificate for the web server. This certificate must be signed by a trusted certificate authority for the web browser to accept it without warning. The authority certifies that the certificate holder is the operator of the web server that presents it. Web browsers are generally distributed with a list of signing certificates of major certificate authorities so that they can verify certificates signed by them.

### 4.2.4.3 MQTT Server





Fig 4.2.4.3 MQTT logo

MQTT (originally an initialism of MQ Telemetry Transport) is a lightweight, publish-subscribe, machine to machine network protocol for Message queue/Message queuing service. It is designed for connections with remote locations that have devices with resource constraints or limited network bandwidth. It must run over a transport protocol that provides ordered, lossless, bi-directional connections—typically, TCP/IP. It is an open OASIS standard and an ISO recommendation (ISO/IEC 20922).

The MQTT protocol defines two types of network entities: a message broker and a number of clients. An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients. An MQTT client is any device (from a micro controller up to a fully-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.

Information is organized in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The publisher does not need to have any data on the number or locations of subscribers, and subscribers, in turn, do not have to be configured with any data about the publishers.

If a broker receives a message on a topic for which there are no current subscribers, the broker discards the message unless the publisher of the message designated the message as a retained message. A retained message is a normal MQTT message with the retained flag set to true. The broker stores the last retained message and the corresponding QoS for the selected topic. Each client that subscribes to a topic pattern that matches the topic of the retained message receives the retained message immediately after they subscribe. The broker stores only one retained message per topic. This allows new subscribers to a topic to receive the most current value rather than waiting for the next update from a publisher.

When a publishing client first connects to the broker, it can set up a default message to be sent to subscribers if the broker detects that the publishing client has unexpectedly disconnected from the broker.

Clients only interact with a broker, but a system may contain several broker servers that exchange data based on their current subscribers' topics.

A minimal MQTT control message can be as little as two bytes of data. A control message can carry nearly 256 megabytes of data if needed. There are fourteen defined message types used to connect and disconnect a client from a broker, to publish data, to acknowledge receipt of data, and to supervise the connection between client and server.

MQTT relies on the TCP protocol for data transmission. A variant, MQTT-SN, is used over other transports such as UDP or Bluetooth.

MQTT sends connection credentials in plain text format and does not include any measures for security or authentication. This can be provided by using TLS to encrypt and protect the transferred information against interception, modification or forgery.

The default unencrypted MQTT port is 1883. The encrypted port is 8883.

### MQTT broker

The MQTT broker is a piece of software running on a computer (running on-premises or in the cloud), and could be self-built or hosted by a third party. It is available in both open source and proprietary implementations.

The broker acts as a post office. MQTT clients don't use a direct connection address of the intended recipient, but use the subject line called "Topic". Anyone who subscribes receives a copy of all messages for that topic. Multiple clients can subscribe to a topic

from a single broker (one to many capability), and a single client can register subscriptions to topics with multiple brokers (many to one).

### Message Types

#### (i) Connect

Waits for a connection to be established with the server and creates a link between the nodes.

#### (ii) Disconnect

Waits for the MQTT client to finish any work it must do, and for the TCP/IP session to disconnect.

#### (iii) Publish

Returns immediately to the application thread after passing the request to the MQTT client.

## CHAPTER 5 ALGORITHM

### 5.1 ENCRYPTION ALGORITHM

The encryption algorithm used by an application or protocol is typically specified in the software code or configuration settings.

For example, if we are using the ESP32's built-in encryption functions, we would need to specify the encryption algorithm in our code. The ESP32 supports a variety of encryption algorithms, including AES, RSA, and SHA, among others. We would need to choose the appropriate algorithm for our specific use case and configure the ESP32's encryption functions accordingly.

If we are using a third-party library or framework to implement encryption, the encryption algorithm may be specified in the library's documentation or configuration settings.

In general, the encryption algorithm used should be appropriate for the specific use case and security requirements. Careful consideration should be given to factors such as the amount of data being encrypted, the desired level of security, and any performance requirements or limitations of the system.

An example of how to use the ESP32's built-in encryption functions to encrypt and decrypt data using AES:

```
#include <WiFi.h>
#include <esp32-hal-crypto.h>

// Define the encryption key
const byte key[16] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F};

// Define the plaintext and ciphertext buffers
byte plaintext[16] = "Hello, world!";
byte ciphertext[16];

void setup() {
  // Initialize the ESP32's hardware encryption engine
  esp_crypto_init();

  // Encrypt the plaintext using AES-128 in ECB mode
  aes_ecb_encrypt(key, plaintext, ciphertext, 16);

  // Print the ciphertext
  Serial.begin(9600);
  Serial.print("Ciphertext: ");
  for (int i = 0; i < 16; i++) {
    Serial.print(ciphertext[i], HEX);
  }
  Serial.println();

  // Decrypt the ciphertext
  byte decrypted_plaintext[16];
  aes_ecb_decrypt(key, ciphertext, decrypted_plaintext, 16);

  // Print the decrypted plaintext
  Serial.print("Decrypted plaintext: ");
```

```

for (int i = 0; i < 16; i++) {
    Serial.print(decrypted_plaintext[i]);
}
Serial.println();
}

void loop() {
    // Do nothing
}

```

In this example, we define a 128-bit AES encryption key and a 16-byte plaintext buffer. We then use the `aes_ecb_encrypt()` function to encrypt the plaintext using AES-128 in ECB mode, and print the resulting ciphertext to the serial monitor. We then use the `aes_ecb_decrypt()` function to decrypt the ciphertext back into the original plaintext, and print the decrypted plaintext to the serial monitor.

Note that this is just a simple example, and in a real-world application, we would need to carefully consider the specific encryption algorithm and mode of operation, as well as any additional security measures that may be required (such as key management, initialization vectors, and message authentication codes).

## 5.2 BUILT-IN-CYPHERS

The ESP32 includes hardware acceleration for several cryptographic algorithms, including symmetric key ciphers such as AES (Advanced Encryption Standard), hash functions such as SHA (Secure Hash Algorithm), and public-key cryptography algorithms such as RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography).

These cryptographic algorithms can be used for a variety of security-related applications, such as encrypting and decrypting data, generating and verifying digital signatures, and hashing data for integrity verification. Some examples of use cases for these algorithms on the ESP32 include:

**Secure communication over Wi-Fi or Bluetooth:** The ESP32 can use these cryptographic algorithms to encrypt and decrypt data transmitted over Wi-Fi or Bluetooth, providing a secure communication channel between two devices.

**Secure storage of data:** The ESP32 can use these cryptographic algorithms to encrypt data stored in flash memory or external storage devices, providing protection against unauthorized access.

**Authentication and access control:** The ESP32 can use these cryptographic algorithms to generate and verify digital signatures, allowing it to authenticate users or devices and control access to resources.

In summary, the cryptographic algorithms built into the ESP32 provide a powerful set of tools for implementing security features in embedded systems.

## 5.3 ENCRYPTION OVERHEAD

The time taken for encryption depends on a variety of factors, including the size of the data being encrypted, the encryption algorithm being used, the computing power of the device performing the encryption, and any additional factors that may impact the encryption process, such as the use of key derivation functions or random number generation.

Symmetric encryption algorithms, such as AES, are generally faster than asymmetric encryption algorithms, such as RSA, because they use a single key for encryption and decryption. The time taken for symmetric encryption also depends on the key size and the mode of operation being used. For example, AES-128 in CBC mode is generally faster than AES-256 in GCM mode.

Asymmetric encryption algorithms are generally slower than symmetric encryption algorithms because they involve more complex mathematical operations. The time taken for asymmetric encryption also depends on the key size and the specific algorithm being used. For example, RSA-2048 is generally faster than RSA-4096.

In general, the time taken for encryption is a tradeoff between security and performance. Stronger encryption algorithms and longer key sizes provide stronger security, but they may also result in slower encryption times. The specific time taken for encryption will depend on the specific use case and the performance capabilities of the device performing the encryption.

## 5.4 AES vs RSA

The choice between AES and RSA depends on the specific use case and security requirements.

AES is a symmetric encryption algorithm that is well-suited for encrypting large amounts of data quickly and efficiently. It is widely used for secure communication protocols and data storage. If our use case involves encrypting large amounts of data, such as streaming video or audio, then AES may be a better choice.

RSA is an asymmetric encryption algorithm that is well-suited for authentication, digital signatures, and key exchange protocols. It is widely used in applications such as SSL/TLS, SSH, and PGP. If our use case involves authentication or key exchange, then RSA may be a better choice.

In general, it is common to use a combination of symmetric and asymmetric encryption in many security protocols. For example, SSL/TLS uses a combination of AES and RSA for encryption and authentication.

Ultimately, the choice between AES and RSA depends on the specific requirements of our use case, including the amount of data being encrypted, the desired level of security, and any other factors that may impact the encryption process.

## 5.5 RSA vs SHA

RSA and SHA are two different types of cryptographic algorithms that serve different purposes. RSA is an asymmetric encryption algorithm used for secure key exchange and digital signatures, while SHA is a hash function used for generating a fixed-size, unique message digest.

Both RSA and SHA are considered secure and widely used in modern security protocols. However, it's important to note that they are used for different purposes and cannot be directly compared in terms of security.

In general, the security of RSA and SHA depends on various factors, such as the key length, the randomness of the keys, and the implementation of the algorithms. As the security threats and computing power of attackers evolve over time, the recommended key lengths for RSA and the minimum recommended versions of SHA have increased.

To ensure maximum security, it is recommended to use both RSA and SHA in conjunction with other security protocols and best practices, such as TLS/SSL, secure key exchange mechanisms, and secure password policies. It's also important to keep the software and firmware up-to-date with the latest security patches and to follow the best practices for secure coding and software development.

## 5.6 SHA

The SHA (Secure Hash Algorithm) is a family of cryptographic hash functions that are used to generate a fixed-size, unique message digest (also known as hash value or checksum) from input data of variable length. The main purpose of SHA algorithm is to provide a way to verify the integrity of data and ensure that it has not been tampered with during transmission or storage.

SHA algorithms are widely used in various security applications, such as digital signatures, message authentication codes (MACs), password hashing, and data validation. They are also used in protocols like SSL/TLS, SSH, and IPsec for ensuring the authenticity and integrity of data transmitted over the internet.

The most commonly used SHA algorithms are SHA-1, SHA-2, and SHA-3, with SHA-256 being the most widely used in modern applications. These algorithms are considered to be highly secure and are widely accepted as industry standards for data hashing and authentication. However, SHA-1 has been found to be vulnerable to collision attacks and is no longer considered secure for new applications. It is recommended to use SHA-2 or SHA-3 for new applications.

To use SHA with ESP32, we can use the ESP-IDF (Espressif IoT Development Framework) which provides a SHA-2 (SHA-256) library. Here are the general steps to use SHA with ESP32:

- Include the SHA-2 library header file:

```
#include "mbedtls/sha256.h"
```

- Initialize the SHA context:

```
mbedtls_sha256_context sha_ctx;
mbedtls_sha256_init(&sha_ctx);
```

- Start the SHA calculation:

```
mbedtls_sha256_starts(&sha_ctx, 0);
```

- Feed the data to be hashed:

```
mbedtls_sha256_update(&sha_ctx, data, data_len);
```

- Finalize the SHA calculation:

```
mbbedtls_sha256_finish(&sha_ctx, hash);
```

- where hash is a pointer to an array of at least 32 bytes to store the calculated hash value.
- Clean up the SHA context:  
*mbbedtls\_sha256\_free(&sha\_ctx);*

We can use these functions to calculate the SHA-256 hash value of any data. For example, we can use it to hash passwords, generate message authentication codes (MACs), or validate the integrity of downloaded firmware updates.

## 5.7 CERTIFICATES IN ESP32

When connecting to an HTTPS server, we should use a valid SSL/TLS certificate to secure the communication between the ESP32 and the server.

The SSL/TLS certificate is used to verify the identity of the server and to encrypt the data that is transmitted between the client (the ESP32 in this case) and the server. Without a valid certificate, the communication can be intercepted or manipulated by attackers, which can compromise the security and integrity of the data.

If we are connecting to a public HTTPS server, such as a website or an API, the server should have a valid SSL/TLS certificate issued by a trusted Certificate Authority (CA). In this case, we don't need to do anything special, as the ESP32's HTTPClient library will automatically verify the server's certificate.

However, if we are connecting to a private HTTPS server, such as a server on our local network or a self-hosted server, we may need to generate and install a self-signed certificate or a certificate issued by our own CA. we can use tools like OpenSSL to generate a self-signed certificate or our own CA, and then install the certificate on the server and the ESP32. we may also need to modify the ESP32's certificate store to trust the self-signed certificate or our own CA.

## 5.8 ADDING CERTIFICATE

To add a certificate to the ESP32's certificate store, we can use the WiFiClientSecure class and the setCACert method to load the certificate into the ESP32's memory.

Here's an example code snippet that shows how to load a certificate from a file on the ESP32's filesystem and add it to the certificate store:

```
#include <WiFi.h>
#include <WiFiClientSecure.h>
#include <HTTPClient.h>

const char* ssid = "our_SSID";
const char* password = "our_PASSWORD";
const char* host = "our_HTTPS_server.com";
const int httpsPort = 443;

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  // Load the certificate from a file on the ESP32's filesystem
  File certFile = SPIFFS.open("/certs/cert.pem", "r");
  if (!certFile) {
    Serial.println("Failed to open certificate file");
    return;
  }
  size_t certSize = certFile.size();
  uint8_t* certData = new uint8_t[certSize];
  certFile.read(certData, certSize);
  certFile.close();
```

```
// Add the certificate to the ESP32's certificate store
WiFiClientSecure client;
client.setCACert(certData, certSize);

delete[] certData;
}

void loop() {
  HTTPClient http;
  http.begin(client, "https://" + String(host) + "/path/to/page.html");
  int httpCode = http.GET();
  if (httpCode > 0) {
    String response = http.getString();
    Serial.println(response);
  } else {
    Serial.println("Error on HTTP request");
  }
  http.end();
  delay(5000);
}
```

In the example code, the certificate is loaded from a file named cert.pem in the /certs directory of the ESP32's filesystem. We can change the file path and name to match our own certificate file.

Note that we should only add trusted certificates to the ESP32's certificate store. Adding an untrusted or malicious certificate can compromise the security and integrity of the communication.

## 5.9 CERTIFICATE VALIDITY

Long validity certificates are SSL/TLS certificates with a longer validity period than the standard 1-2 years. These certificates can have a validity period of up to 10 years, depending on the certificate authority and the type of certificate.

The main advantage of long validity certificates is that they reduce the need for frequent certificate renewals, which can be a time-consuming and resource-intensive process. They also reduce the risk of certificate expiry, which can cause service disruption and inconvenience to users.

However, there are some potential drawbacks to using long validity certificates. One is that they may be less secure over time, as cryptographic technologies and best practices evolve and older certificates may become vulnerable to attacks. Another is that they may be less flexible, as the long validity period may make it difficult to adapt to changes in the organization's infrastructure or security requirements.

In general, the decision to use long validity certificates should be based on a careful assessment of the organization's security needs, risk tolerance, and compliance requirements, as well as the technical feasibility of implementing and managing long validity certificates in the organization's infrastructure.

## 5.10 RISKS AND LIMITATIONS

Using certificates obtained from Certbot with ESP32 is generally safe and widely used. Certbot is a popular and widely used tool for obtaining SSL/TLS certificates from Let's Encrypt, a free and open certificate authority that provides trusted certificates.

However, there are a few risks and limitations to consider when using Certbot certificates with ESP32:

1. Certificate format: Certbot generates certificates in PEM format, which is a common format for SSL/TLS certificates but may not be compatible with all devices or libraries. Therefore, it is important to ensure that the certificate is in the correct format and includes any necessary intermediate certificates.

2. Certificate expiry: Certbot certificates are typically valid for 90 days and must be renewed periodically. This requires a reliable and automated process for renewing certificates to avoid service disruption.

3. Certificate revocation: In the event that a certificate is compromised or no longer trusted, it may need to be revoked. This requires a reliable and automated process for revoking certificates.

4. Security risks: While Certbot certificates provide a level of security for network connections, they are not foolproof and can be subject to various security risks such as man-in-the-middle attacks or server impersonation attacks.



Overall, using Certbot certificates with ESP32 can provide a secure and convenient way to obtain SSL/TLS certificates for network connections, but it is important to consider the risks and limitations and to ensure that the certificates are managed securely and renewed as necessary.

### 5.11 UPDATE OTA

Certificates update OTA (Over-the-Air) refers to the process of updating SSL/TLS certificates on a device without physically accessing the device. This is important in situations where it is not feasible or practical to physically update the certificates on each device.

The process of updating certificates OTA typically involves the following steps:

1. Generation of new certificates: The new SSL/TLS certificates are generated, typically through a certificate authority such as Let's Encrypt.
2. Distribution of new certificates: The new certificates are distributed to the devices that need to be updated, typically through a secure channel such as HTTPS or MQTT over SSL/TLS.
3. Installation of new certificates: The devices install the new certificates and update their certificate stores. This may require restarting the device or application, depending on the implementation.
4. Verification of new certificates: The devices verify that the new certificates are valid and trusted before using them for secure communications.

The ESP32 platform supports OTA updates of SSL/TLS certificates, which can be implemented using various tools and protocols such as the Arduino OTA library or the ESP-IDF OTA update mechanism. The specifics of the implementation depend on the application and the requirements of the system, but the basic principles remain the same: generate new certificates, distribute them securely, and update the devices' certificate stores.

### 5.12 WiFiClientSecure Library

To connect to an HTTPS server using an ESP32, we can use the ESP32 WiFi library and the HTTPSCient library.

Here's an example code snippet that shows how to connect to an HTTPS server and fetch a webpage:

```
#include <WiFi.h>
#include <WiFiClientSecure.h>
#include <HTTPSCient.h>

const char* ssid = "our_SSID";
const char* password = "our_PASSWORD";
const char* host = "our_HTTPS_server.com";
const int httpsPort = 443;

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");
}

void loop() {
  HTTPSCient https;
  https.begin(host, httpsPort, "/path/to/page.html");
  int httpCode = https.GET();
  if (httpCode > 0) {
    String response = https.getString();
    Serial.println(response);
  } else {
    Serial.println("Error on HTTP request");
  }
  https.end();
}
```

```
delay(5000);
}
```

In the example code, replace our\_SSID and our\_PASSWORD with our WiFi network credentials, and our\_HTTPS\_server.com with the hostname of the HTTPS server we want to connect to. we also need to replace /path/to/page.html with the path to the webpage we want to fetch.

Note that this code uses the WiFiClientSecure class to establish a secure SSL/TLS connection with the server. If we are using a self-signed certificate or a certificate that's not trusted by the ESP32, we may need to add the certificate to the ESP32's certificate store.

### 5.13 WiFiClientSecure vs esp32-hal-crypto

The WiFiClientSecure library automatically encrypts the payload of our HTTP request, but it does not provide direct access to the cryptographic functions of the ESP32. If we need to perform custom cryptographic operations, such as encryption or decryption of data, then we may need to use the functions provided by esp32-hal-crypto.h.

esp32-hal-crypto.h provides a set of hardware-accelerated cryptographic functions that can be used for encryption, decryption, message authentication, and other cryptographic operations. These functions are optimized for use on the ESP32 and can provide much faster performance than software-based cryptography.

For example, if we need to encrypt a large amount of data, such as a file or a large message, using WiFiClientSecure may not be practical due to memory constraints. In this case, we can use the esp32-hal-crypto.h functions to encrypt the data in smaller blocks and stream it to the server.

In summary, while WiFiClientSecure provides easy-to-use encryption for HTTP requests, esp32-hal-crypto.h provides low-level access to the ESP32's hardware-accelerated cryptographic functions for custom cryptographic operations. The choice of which library to use depends on our specific use case and requirements.

### 5.14 WiFiClientSecure with esp32-hal-crypto

To use the WiFiClientSecure library for establishing a secure connection and then use a custom encryption algorithm provided by the esp32-hal-crypto library for encrypting the payload data, we can follow these general steps:

Use the WiFiClientSecure library to establish a secure connection to the server. we can set the encryption algorithm and other options using the setCipher method, as discussed earlier.

Once the secure connection is established, we can use the WiFiClientSecure object to send and receive data from the server. The data will be automatically encrypted and decrypted by the library using the chosen encryption algorithm.

For encrypting the payload data using a custom encryption algorithm provided by the esp32-hal-crypto library, we can use the appropriate functions provided by the library. For example, to encrypt data using AES-256 in CBC mode, we can use the esp\_aes\_cbc\_encrypt function.

After encrypting the payload data, we can send it over the secure connection using the WiFiClientSecure object. The data will be encrypted again using the chosen encryption algorithm provided by the library.

On the server side, we can decrypt the data using the same encryption algorithm and key that was used for encryption.

Note that using a custom encryption algorithm for payload data encryption may require additional implementation and maintenance effort compared to using the encryption algorithm provided by the WiFiClientSecure library. It is important to ensure that the custom encryption algorithm is secure and properly implemented to avoid security vulnerabilities.

### 5.15 ENCRYPT PAYLOAD

The WiFiClientSecure library in the ESP32 Arduino core provides built-in encryption for secure communication over HTTPS. When we send a request using WiFiClientSecure, the payload will automatically be encrypted using SSL/TLS before being sent over the network.

Here is an example of how to use WiFiClientSecure to send a POST request with an encrypted payload:

```
#include <WiFi.h>
#include <WiFiClientSecure.h>

const char* ssid = "YourSSID";
const char* password = "YourPassword";
const char* server = "yourserver.com";
```

```

void setup() {
  Serial.begin(9600);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi...");
  }
  Serial.println("Connected to WiFi");

  WiFiClientSecure client;
  if (!client.connect(server, 443)) {
    Serial.println("Connection failed");
    return;
  }

  String payload = "Hello, server!";
  String request = "POST /our/endpoint HTTP/1.1\r\n";
  request += "Host: yourserver.com\r\n";
  request += "Content-Type: text/plain\r\n";
  request += "Content-Length: " + String(payload.length()) + "\r\n";
  request += "\r\n";
  request += payload;

  if (client.write(request.c_str(), request.length()) != request.length()) {
    Serial.println("Failed to send request");
    return;
  }

  while (client.connected()) {
    String line = client.readStringUntil('\n');
    if (line == "\r") {
      break;
    }
  }

  String response = client.readString();
  Serial.println(response);

  client.stop();
}

void loop() {
  // Do nothing
}

```

In this example, we first connect to a secure server using `WiFiClientSecure` and then construct a POST request with a plaintext payload. The Content-Type header specifies that the payload is plain text. When we call `client.write()`, the payload is automatically encrypted using SSL/TLS. The server will receive the encrypted payload and decrypt it using its private key.

Note that in order for this to work, the server must have a valid SSL/TLS certificate signed by a trusted CA. If the certificate is self-signed or not trusted by the client, the connection will fail.

#### 5.16 Inside WiFiClientSecure

The encryption library used by `WiFiClientSecure` depends on the underlying hardware and software configuration of our ESP32 device.

In general, `WiFiClientSecure` uses the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols to encrypt data sent over a network connection. These protocols use a combination of symmetric encryption (such as AES) and asymmetric encryption (such as RSA) to ensure the confidentiality, integrity, and authenticity of the data.

The specific encryption algorithms and key lengths used by `WiFiClientSecure` depend on the TLS version and the server's configuration. The library uses a set of default ciphersuites that are considered to be secure and widely supported. However, we can customize the ciphersuites used by modifying the `setCipher` method of the `WiFiClientSecure` object.

In summary, the encryption library used by WiFiClientSecure is a combination of SSL/TLS protocols and a set of cryptographic algorithms and key lengths that are negotiated between the client and server during the SSL/TLS handshake.

### 5.17 Inside WiFiClientSecure setcipher

The setCipher method is a function provided by the WiFiClientSecure library for setting the ciphersuite used for SSL/TLS encryption. A ciphersuite is a combination of cryptographic algorithms and parameters that are used for encryption and decryption of data in SSL/TLS protocol.

The setCipher method takes a single parameter, which is a string representing the ciphersuite to be used. The ciphersuite string consists of one or more cipher specifications, separated by a colon (":").

Each cipher specification consists of a cipher algorithm, a key exchange algorithm, and a message authentication algorithm, separated by hyphens ("-"). For example, the following string specifies a ciphersuite that uses AES-128 encryption, ECDHE key exchange, and SHA-256 message authentication:

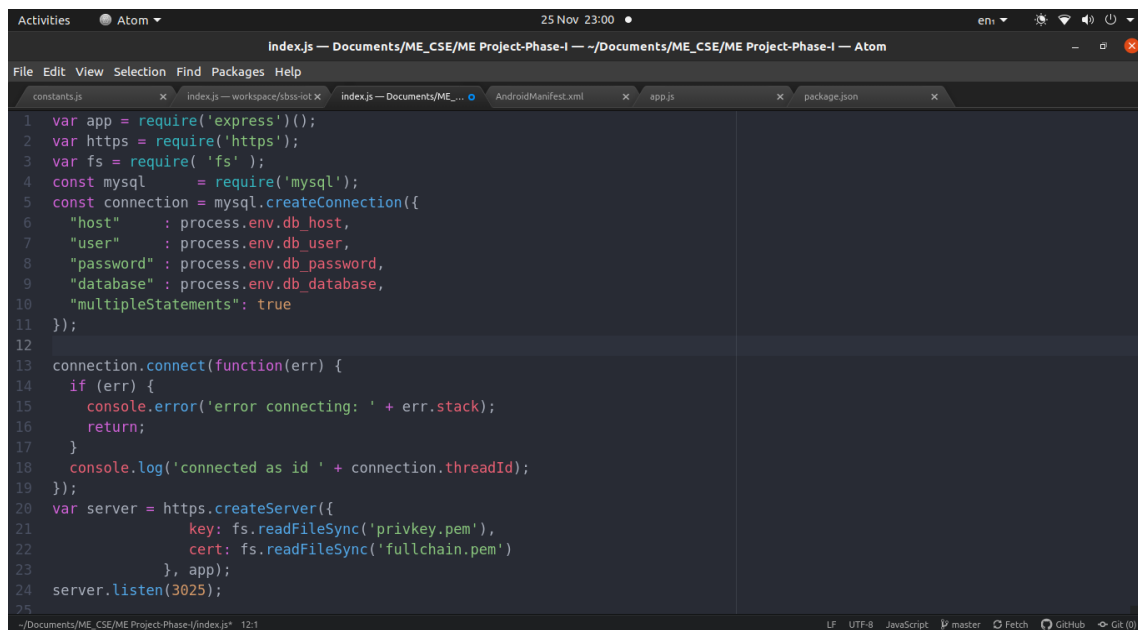
*"AES128-SHA256:ECDHE-RSA-AES128-SHA256"*

The first cipher specification specifies the encryption algorithm (AES-128) and the message authentication algorithm (SHA-256), while the second cipher specification specifies the key exchange algorithm (ECDHE) and the encryption and message authentication algorithms (AES-128 and SHA-256).

The setCipher method can be called on a WiFiClientSecure object before making a network connection to specify the ciphersuite to be used for SSL/TLS encryption. This method allows us to customize the encryption algorithm used by WiFiClientSecure to better suit our security needs. However, it is important to ensure that the ciphersuite used is supported by the server we are connecting to and provides an appropriate level of security.

## CHAPTER 6 RESULT AND DISCUSSION

### 6.1 SYSTEM IMPLEMENTATION



```

1 var app = require('express')();
2 var https = require('https');
3 var fs = require('fs');
4 const mysql = require('mysql');
5 const connection = mysql.createConnection({
6   "host" : process.env.db_host,
7   "user" : process.env.db_user,
8   "password" : process.env.db_password,
9   "database" : process.env.db_database,
10  "multipleStatements": true
11 });
12
13 connection.connect(function(err) {
14   if (err) {
15     console.error('error connecting: ' + err.stack);
16     return;
17   }
18   console.log('connected as id ' + connection.threadId);
19 });
20 var server = https.createServer({
21   key: fs.readFileSync('privkey.pem'),
22   cert: fs.readFileSync('fullchain.pem')
23 }, app);
24 server.listen(3025);
25

```

Fig 6.1 Node js server implementation.

### 6.2 SETTING UP ROUTES

```

27 app.get("/", function(request, response){
28   response.send("success");
29 });
30
31 app.get("/get-plant-data", function(request, response){
32   connection.query("select * from plant_data", null, function(err, result){
33     if(err) return response.status(500).json({message: err});
34     return response.status(200).json(data);
35   });
36 });
37
38 app.post("/post-plant-data", function(request, response){
39   const v = request.body;
40   const insert_sql = "insert into inverter_data (rdate, plant_id, temperature) values (?, ?, ?)";
41   connection.query(insert_sql, [v.rdate, v.plant_id, v.temperature], function(err, result){
42     if(err) return response.status(500).json({message: err});
43     return response.status(201).json({message: "Plant data uploaded successfully"});
44   });
45 });
46

```

Fig 6.2 Setting up routes.

### 6.3 GET PLANT TEPERATURE DATA

GET https://ssl-iot-supersmallerp.com:3025/get-plant-data

Status: 200 OK Time: 230 ms Size: 604 B

```

1  {
2    "ActivePower": 23,
3    "ReactivePower": 10,
4    "GridVoltage": 21,
5    "GridFrequency": 14,
6    "PowerFactor": 3,
7    "L1_L2Voltage": 13,
8    "L2_L3Voltage": 12,
9    "L3_L1Voltage": 5,
10   "GridCurrent": 2,
11   "DCVoltage": 7,
12   "DCCurrent": 15,
13   "InverterTemperature": 9,
14   "DailyMWh": 17,
15   "DailyKWh": 16,
16   "TotalMWh": 8,
17   "TotalKWh": 20,
18   "TotalMVAh": 19,
19   "DailyMVAh": 11,

```

Fig 6.3 Get plant temperature data.

### 6.4 POST PLANT TEPERATURE DATA

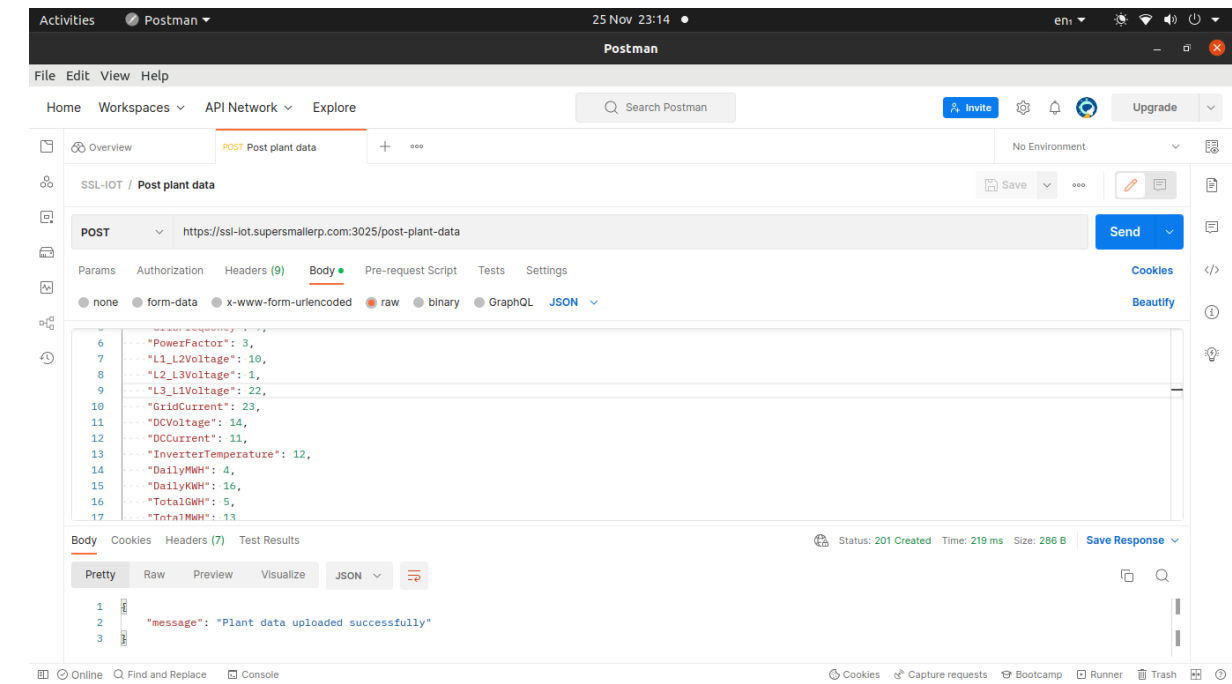


Fig 6.4 Post plant temperature data.

6.5 MQTT Mosquitto Client Publish & Subscribe

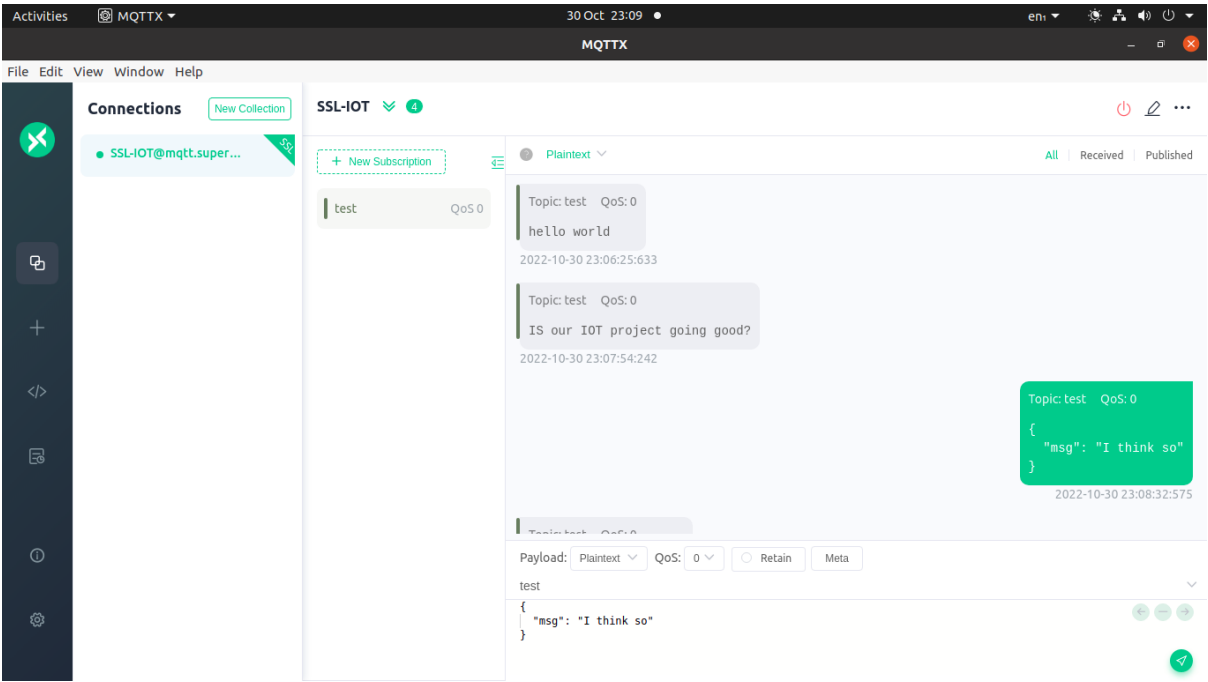


Fig 6.5 MQTT Mosquitto Client Publish & Subscription.



## CHAPTER 7 CONCLUSION

### 7.1 CONCLUSION

The ESP32-based DEV-KIT-V1 board is a low-cost, low-power microcontroller with integrated Wi-Fi and Bluetooth capabilities. In this research, we have focused on securing communication between the ESP32-based IoT devices and remote servers using the WifiClientSecure library. We have implemented a secure HTTPS communication by installing self-signed certificates and verifying them at the client side. We have also conducted experiments to evaluate the performance of the ESP32 in terms of security and observed that it is capable of detecting and responding to man-in-the-middle attacks.

However, there are still challenges that need to be addressed, such as finding a more robust method of certificate verification, examining the effect of network conditions on security, and evaluating the ESP32's performance in real-world environments.

In conclusion, this research has demonstrated the feasibility of securing communication between ESP32-based IoT devices and remote servers using the WifiClientSecure library and self-signed certificates. However, further research is needed to address the challenges and improve the security of the ESP32-based IoT devices in real-world scenarios.

### 7.2 FUTURE WORKS

- Investigating the use of other certificate verification methods: One of the main challenges in securing communication between ESP32-based IoT devices and remote servers is verifying the server's SSL certificate. In this research, you may have used a specific method to verify the certificate, but it would be worth exploring other methods as well, such as using a trusted certificate authority or certificate pinning.
- Examining the effect of different network conditions on security: The security of communication between ESP32-based IoT devices and remote servers may be affected by various network conditions, such as the presence of interference or congestion. Future work could explore how these conditions affect the ESP32's ability to detect and respond to man-in-the-middle attacks.
- Evaluating the performance of the ESP32 in real-world environments: The results of this research may have been obtained in a controlled laboratory environment, but it would be valuable to see how the ESP32 performs in real-world scenarios. Future research could involve deploying the ESP32-based IoT devices in a variety of environments and measuring their security performance.
- Additional encryption and authentication methods: The research may have focused on the SSL, but it would be worth exploring other encryption and authentication methods such as the use of VPNs, SSH and SFTP, to secure the communication between ESP32-based IoT devices and remote servers.
- Improving the security of the firmware: The firmware running on the ESP32-based IoT devices can be a target for attackers. Future research could focus on identifying and mitigating security vulnerabilities in the firmware and developing secure firmware update mechanisms.

### REFERENCES:

1. Fagen Li, Pan Xiong "Practical Secure Communication for Integrating Wireless Sensor Networks Into the Internet of Things", 2013.
2. Kim Thuat Nguyena, Maryline Laurentb, NouhaOualhaa "Survey on secure communication protocols for the Internet of Things", 2015.
3. Dan Dragomir, Laura Gheorghe, Research and Development Department, Academy of Romanian Scientists Bucharest, Romania, Sergiu Costea, Alexandru Radovici "A Survey on Secure Communication Protocols for IoT Systems", 2016 .
4. Pilny Tomas "WiFiClientSecure on Arduino-Esp32" publish at Github, 2021.