Alexander Matias Timko

Cybersecurity of Internet of Things Devices: A Secure Shell Implementation

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

5 May 2022

Abstract

Author:	Alexander Matias Timko
Title:	Cybersecurity of Internet of Things Devices: A Secure Shell Implementation
Number of Pages:	50 pages + 1 appendix
Date:	5 May 2022
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	IoT and Cloud Computing
Supervisors:	Erik Pätynen, Senior Lecturer

The objective of this thesis was to provide a theoretical overview of the current cybersecurity climate concerning the IoT and IoT devices, analyse both wireless and communication protocols used in the IoT, and research the viability of implementing an SSH architecture into an IoT system as a means of securing data in transit.

The primary aim of the implementation was to create an IoT system which would utilise an SSH architecture through the Secure Copy Protocol to transfer data between sensor node and edge device. The sensor node, acting as SSH Client, consisted of a NodeMCU ESP32 microcontroller with BMP180 pressure and temperature sensor and DHT-11 humidity and temperature sensor. The edge device, acting as SSH server, was a computer running an Ubuntu operating system, which would collect transmitted sensor node data and visualize the data in a graph. Furthermore, the secondary aim of the implementation was to use the Secure Copy Protocol to allow secure Over-the-Air firmware updates of the sensor node.

Further development and research are needed to apply the implementation to a wider range of IoT devices, test processing and power implications of the implementation, and develop a secondary security mechanism for Over-the-Air updates.

However, the results of the implementation showed that using an SSH Client/Server architecture in an IoT system is a viable means of securing and encrypting transmitted sensor and firmware update data between an IoT sensor node and IoT edge device.

Keywords: cybersecurity, IoT, SSH, SCP, ESP-32 microcontroller, OTA updates

Contents

List of Abbreviations

1 Introduction 1

2 Technical Background 3

2.1 Cybersecurity of IoT and IoT Devices 3
2.1.1 IoT Threats and Mitigations 4
2.2 IoT Protocols 6
2.3 Secure Shell Protocol (SSH) 8
2.3.1 SSH Architecture 8
2.3.2 Encryption 10
2.3.3 Authentication 10
2.3.4 Secure Copy Protocol (SCP) 11
2.3.5 Disadvantages and Advantages of SSH 13
2.3.6 SSH in the IoT 14
2.4 Over-the-Air (OTA) Updates 15
2.4.1 Benefits of OTA Updates in the IoT 15
2.4.2 Concerns of OTA Updates in the IoT 16
2.5 IoT Devices 17
2.5.1 ESP-32 NodeMCU Microcontroller 18
2.5.2 Sensors 20

3 Implementation 22

3.1 Implementation of Secure Shell Copy 22
3.1.1 Key Pair Generation 23
3.1.2 Setting Up SSH Server and SCP Client 26
3.1.3 Including BMP Sensor: Collecting and Sending Data via SCP 30
3.1.4 Visualization of Sensor Data 36
3.2 Implementation of OTA Updates 38

4 Results 43

5 Discussion 45

5.1 Limitations 45
5.2 Future Development 46

6 Conclusion 47

References 48

Appendices
Appendix 1: Sensor Task Function

List of Abbreviations

IoT: Internet of Things

SCP: Secure Copy Protocol

SSH: Secure Shell

OTA: Over-the-Air

DOS: Denial-of-Service

DDOS: Distributed Denial-of-Service

DNS: Domain Name System

OWASP: Open Web Application Security Service

3G: Third Generation

4G: Fourth Generation

5G: Fifth Generation

LPWAN: Low Power Wide Area Network

LoRaWAN: "Long Range" Wide Area Network, IoT communication protocol and system architecture.

LoRa: "Long Range" communication link for LoRaWAN networks.

XMPP: Extensible Messaging and Presence Protocol

CoAP: Constrained Application Protocol

MQTT:	Message Queuing Telemetry Transport
SSL/TLS:	Secure Sockets Layer / Transport Layer Security
DTLS:	Datagram Transport Layer Security
TCP:	Transmission Control Protocol
UDP:	User Datagram Protocol
MITM:	Man-in-the-Middle Attack
MAC:	Message Authentication Codes
GPIO:	General-Purpose Input/Output
SoC:	System on a Chip
SPI:	Serial Peripheral Interface
SPIFFS:	Serial Peripheral Interface Flash File System
I2C:	Inter-Integrated Circuit
SFTP:	Secure File Transfer Protocol
SRAM:	Static-Random-Access Memory
ROM:	Read-Only Memory
IEEE:	Institute of Electrical and Electronics Engineers
DSA:	Digital Signatures Algorithm
RSA:	Rivest-Shamir-Adleman algorithm

1 Introduction

The Internet of Things (IoT), as of 2022, has become ubiquitous in our lives by creating a network of physical devices on a global scale. These devices are adept to sensing and reacting to the environment they are in, and can communicate with one another, other machines, and computers, by transmitting various forms of data over the internet. Having this data available at any time is what makes it possible for organizations, from varying industries, to save costs, improve revenue, and automate tasks. Additionally, data availability from IoT devices can improve the quality of life for many, from monitoring one's health to reducing one's carbon footprint. However, the constant introduction of IoT devices to the global network of connected IoT devices has opened a plethora of attack vectors for malicious threat actors to take advantage, manipulate, and use data to disturb the operations of organizations and threaten the lives of the public.

The purpose of this thesis is to research the viability of implementing the Secure Copy Protocol (SCP) as a way of securing the transmission of data between an IoT device and destination machine. Furthermore, it will investigate the use of Over-the-Air updates using SCP as a solution for updating hard to reach IoT devices in a secure manner.

Firstly, the paper will focus on creating an understanding of the current cybersecurity climate of IoT and IoT devices, by observing its' threats and threat mitigation techniques. Followed by providing the reader with a technical overview of the Secure Shell Protocol (SSH), the advantages and disadvantages of OTA updates, and a synopsis of the technical specifications of the IoT device and sensor used for this thesis.

Secondly, the implementation section of the paper will describe how SCP was applied to transmitting sensor data securely from microcontroller to computer and creating a visual representation of the data. Then outline how OTA updates can be performed securely from computer to microcontroller. Thirdly, an

investigation on the results of the implementation and analysis of whether the data being transferred has been secured. Afterward, a discussion of the drawbacks of the research conducted and future developments will be made.

Lastly, the conclusive section of the thesis will give a summary of the viability of implementing SCP as a means of securing data in transit and OTA updates. The aim of this thesis is to provide awareness of the current state of cybersecurity of IoT and allow the reader to reproduce and implement SCP to transmit sensor data, visualize the data, and apply secure OTA updates of IoT devices.

2 Technical Background

The following section will provide an overview of the current cybersecurity state of IoT and IoT devices, as well as common threats and mitigation techniques concerning IoT. Subsequently, a brief analysis of commonly used IoT data communication protocols, followed by an in-depth examination of the SSH protocol and reasoning as to why SCP would be a viable option as a data communication protocol for IoT devices.

The two succeeding sections will focus on the purpose, benefits, and pitfalls of OTA updates. Moreover, this section of the paper will introduce common IoT devices and provide a technical review of the ESP-32 microcontroller and BME280 sensor used for this thesis.

2.1 Cybersecurity of IoT and IoT Devices

Since the inception of the term “Internet of Things” in a presentation at MIT by Kevin Ashton in 1999, the world has seen an incremental increase of the number of connected IoT devices. It is estimated that by the year 2025 the number of devices will climb to 38.6 billion and is projected to rise to 50 billion by 2030. Adding to this continually increasing number of devices only opens novel attack vectors for threat actors to take advantage of, introducing new threats to the environments and networks these devices operate it. It has become a common saying that the Internet of Things has come to be the “Internet of Things to hack”. [1; 2.]

IoT has evolved over the past two decades to be involved and integrated into countless cyber-physical systems. Systems related to smart homes, smart hospitals, smart cities, industrial supply chains and manufacturing, smart power grids to name a few. [3.] As IoT has spread through many domains, both public and private, there has been a focus on the availability data rather than its’ confidentiality and integrity. Traditionally, network devices placed higher value on confidentiality, then integrity, and then availability, which meant it was easier

to implement more robust security systems. [1.] Paying more attention to data availability has brought a certain challenge as to how to IoT systems are implemented and has caused a divide in how IoT devices are manufactured making it difficult to standardize IoT security. Difficulty in standardizing the manufacture of IoT devices and the protocols used for IoT data communication has enabled threat actors to discover potential vectors into disrupting IoT devices and their networks. However, as the number of potential threats has increased so has the research conducted towards mitigating these threats. [2.] The following subsection will provide a few examples of the threats IoT faces and the available mitigation techniques.

2.1.1 IoT Threats and Mitigations

The reason why threat actors find IoT devices as an enticing means of causing some form of cyber attack is because of what the threat actor can achieve by accessing an IoT device. Depending on the intention of the threat actor, cyber attacks, or malicious intents through IoT devices can take many forms. For example, threat actors have used IoT devices to cause disruptions in the shape of Denial-of-Service (DOS) of Distributed-Denial-of-Service (DDOS) attacks. One instance of such a DDOS attack took place when attackers took advantage of an insecure operating system running on hundreds of thousands of IoT devices and sent Domain Name System (DNS) queries to a DNS provider named Dyn, which caused the disruption of services like Twitter and Netflix. [2; 4.] Another IoT DDOS attack occurred in 2016, when threat actors focused their attack on an Internet journalist, Brian Kreb, and his website, preventing users from accessing it. The two DDOS examples described were achieved by what is now known known as the Mirai worm botnet, where threat actors were able to infect IoT devices because of a vulnerability with default login credentials. [1; 5.]

Another threat the IoT faces is what people or organizations consider to be an IoT device, some devices, such as network printers, can be overlooked. In 2017, an individual used a program to crawl through the Internet to discover vulnerable printer making them print out taunting messages, over 150,000

printers were hacked around the world. This type of attack can be seen more as vandalism than anything else, however, there can be serious future implications of this type of cyber attack. For example, attacking bio printers, even though they are in the early stages in the medicinal field, could cause devastating problems for patients who would depend on them. [6.]

It can be discerned that the largest threat to the IoT is the vulnerabilities found in IoT devices, the Open Web Application Security Project (OWASP) released a list of the top ten vulnerabilities which the IoT faced in 2018. Even though released in 2018, they are still valid today, to name a few, weak or default passwords and settings, insecure data transfer and storage, deficiency of secure update methods and device management, or inadequate privacy protection. [7.] These vulnerabilities can also be found easily using both simple and advanced tools. For example, network protocol scanning tools, such as Nmap or Zmap, internet tools, like the Shodan IoT search engine which allows users to find any devices that are open to the internet, or penetration testing kits, like Burpsuite or Metasploit. [5; 7.]

Even though it seems like that threat actors may have the upper hand there are many ways one can mitigate the threats faced by the IoT. Simple approaches, such as changing default passwords and configurations can make a huge difference in securing IoT devices from attacks. Researchers have also found various ways to mitigate threats, for example through the use if Intrusion Detection Systems (IDS) which have been particularly excellent at detecting DDOS attacks from IoT devices and have shown that they can prevent the exploitation of IoT devices found internal networks. [2; 4.] Moreover, considerations made towards the communication protocols used and the implementation of OTA updates within an IoT system can have a significant effect on alleviating threats. The following section of this paper will examine the common wireless and data communication protocols used in the IoT devices.

2.2 IoT Protocols

The type of protocols used within the IoT architectures fall into two categories, wireless protocols, and communication protocols. Wireless protocols allow IoT devices to connect to the Internet or to connect to one another and operate in either the Physical or Data Link layer of the protocol stack. For example, Global System for Mobile Communications (GSM), 3G, 4G, and 5G are forms of wireless communication which use cellular networks and are used for long to short range communication. Wi-Fi, also known as the IEEE 802.11 protocol, is another example of wireless protocol used in home and office networks for many types of devices, ranging from laptops, televisions to IoT devices, providing mid-range communication. [1.]

Furthermore, wireless IoT protocols developed for low power wide area networks (LPWAN) include the Zigbee, SigFox, and IEEE 802.15.4g protocols, and the LoRaWAN IoT protocol and system architecture, which uses the LoRa communication link, have become increasingly popular. These wireless protocols are used specifically for IoT devices and is especially convenient for applications which have low power requirements. [8.] Finally, Bluetooth and Z-Wave are further examples of wireless protocols used in IoT, the drawback of Zigbee, Bluetooth and Z-Wave is that they cannot interact with the Internet directly and must use gateway devices. [1.]

Communication protocols in the IoT operate in the Application layer of the IoT protocol stack, and are used to communicate application-level data, meaning they connect the data collected by the IoT device to the end user. There are many available IoT communication protocols to choose from and are selected based on their applicability, security, deployment, and scalability. The three following protocols are the commonly used communication protocols used in the IoT.

Extensible Messaging and Presence Protocol (XMPP) is a real-time protocol which allows users to communicate with IoT devices using instant messaging

services. XMPP uses Secure Socket Layer / Transport Layer Security (SSL/TLS) to encrypt the data which is passed between devices and uses the Transmission Control Protocol (TCP) to ensure that data packets are being sent to the receiving device. A disadvantage of XMPP is that it does not provide end-to-end encryption of data and is not the best suited for machine-to-machine (M2M) communication. [9.]

Constrained Application Protocol (CoAP) is another IoT communication protocol used for M2M applications. CoAP uses a simple request/response style of communications following a client-server model, and uses GET, POST, PUT, and DELETE requests to control the data being transferred. CoAP uses the User Datagram Protocol (UDP) as a way transporting data packets, which means that communication is fast as UDP is a connectionless protocol. The downside of being a connectionless protocol is that there is no insurance that the packets are reaching their destination. Another disadvantage of UDP is that it does not use SSL/TLS encryption, however, CoAP uses Datagram Transport Layer Security (DTLS) as a security protocol. [1; 9.]

Message Queuing Telemetry Transport (MQTT) is an IoT protocol which was developed with the idea of making it as lightweight as possible, making it preferrable for low powered devices for its ability to decrease power consumption. Like CoAP, MQTT implements a publish/subscribe model using a client-broker model and uses TCP as its transport layer protocol, thus ensuring data is being sent and received. A disadvantage of using MQTT is, due to the lightweight nature of the protocol, is the lack of data encryption it supports; however, it is possible to implement the SSL/TLS security protocol on top of the TCP connections created by MQTT. [9.]

Deciding on the correct protocols to use within an IoT system depends on the application and outcome one wants to achieve, however, considerations towards the security of the protocols used should always be accounted for. The following section of this paper will provide an overview of the SSH and SCP

protocols, exploring the possible advantages and disadvantages they could provide to an IoT system.

2.3 Secure Shell Protocol (SSH)

After a password-sniffing attack in the Helsinki University of Technology in Finland, Tatu Ylonen decided to develop a secure way of remotely accessing his machine on the university network. In 1995 marked the conception of SSH1, the universal term used for the SSH protocol and software products, and SSH-1, the name of Version 1 of the protocol itself. SSH1 was intended to replace insecure protocols such as Remote Shell (RSH), Remote Copy (RCP) and telnet. By 1996, the next major version, SSH2, of the protocol was proposed by SSH Communication Security Corporation (SCS) and was released in 1998 as a more, secure version of the protocol. Due to its proprietary nature SSH2 wasn’t adopted as widely as the open-source SSH1, however, when OpenSSH began development of their own SSH2 protocol, based on the last free release of SSH1, it became widely adopted and is used by millions of people worldwide. Reference to SSH in this paper will refer to SSH2, as SSH1 has been deprecated and deemed insecure. The following subsection will examine the SSH architecture. [10; 11.]

2.3.1 SSH Architecture

SSH uses a client/server model to form communication between devices, that is, a device connecting to a server must have a SSH client program running and the device running the server must have a SSH server software running on it. It uses a packet-based binary protocol which works on top of TCP which transports the data through the transport layer of the TCP/IP stack. The architecture contains many elements which allow SSH to work, it is out of the scope of this paper to examine all these elements but will focus on the main components which allow its functionality. [10; 11.] The diagram in Figure 1 below illustrates the main elements which comprise the SSH architecture.

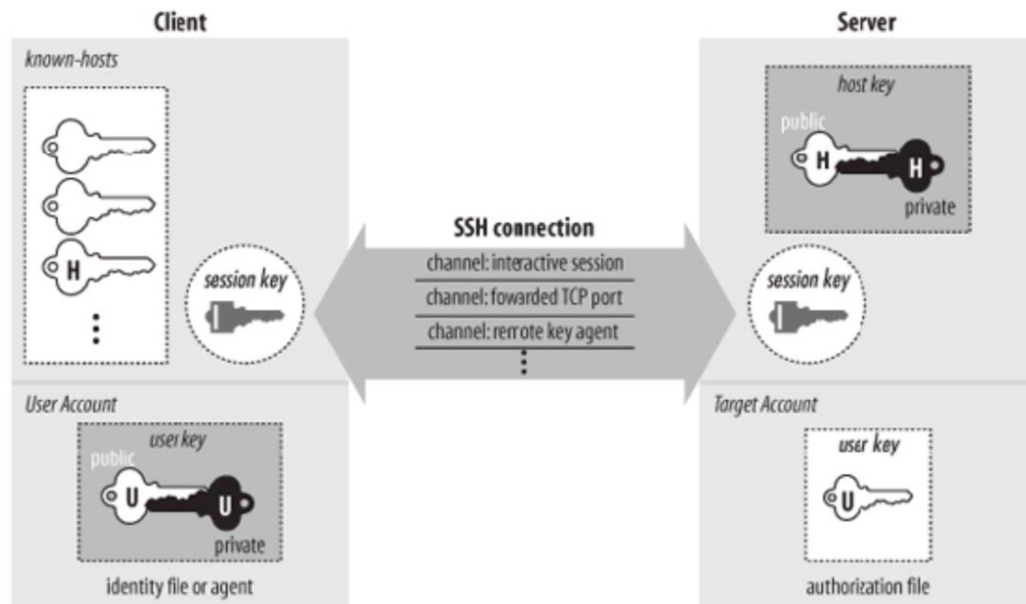


Figure 1. SSH Architecture [10.]

The architecture comprises of two parties, client and server, various types of keys, user, host, and session keys, and the SSH connection. The client is always the initiating party which begins communication with the server by using TCP handshake, which ensures that a TCP socket connection has been created to transmit data between two parties. The server responds by announcing which SSH version is used to make sure both parties are compatible with one another, it also sends its own public host key to authenticate with the client. The client then verifies that the servers public key can be found in it's know-hosts file, a client can have many public server keys as a client can connect to many hosts. If the keys match each other the two parties create a session key, this is created using symmetric encryption, meaning both parties will share the same key, the purpose of the key is to encrypt the current session. After the session has been encrypted the client must use some form of authentication, such as password, public-private key, or host-based authentication. The types of encryptions and authentication methods used through the SSH connection process are discussed in the two subsequent subsections. [10.]

2.3.2 Encryption

There are two forms of encryption which occur during an SSH connection, the first is symmetric and the second is asymmetric. Symmetric encryption happens at the beginning of the SSH session when client and server carry out a key exchange algorithm. This key exchange algorithm occurs after both client and server agree on which algorithm, they will use to create the symmetric key, by agreeing on the algorithm used they will generate identical keys which will be used to encrypt the current SSH session. These identical keys are also known as the session key or shared key, which is unique for every SSH session and are destroyed once the session ends. This type of symmetric encryption is considered secure as the keys are never transmitted between the two ends. [10.]

The second form of encryption used within an SSH connection is asymmetric encryption, this type of encryption, unlike symmetric encryption, uses two different keys for encryption and decryption, these two keys form a public-private key pair. The public key can be known by anyone who uses it, whereas the private key is limited to the machine or individual who holds it and should not be shared. Data is encrypted with the private key and can only be decrypted with the corresponding public key; this ensures that if data reaches a location, it was not intended to it cannot be decrypted. Asymmetric encryption occurs once during the initial phase of the SSH connection, during the key exchange algorithm where they share temporary private keys to create the session key. Asymmetric encryption is used again by the server, by encrypting a challenge for the client using the clients public key to begin the authentication process of the SSH session. [10.]

2.3.3 Authentication

There are a few ways one can authenticate with an SSH server, it is out of scope of this paper to review them all, a focus will be made on the two most popular authentications methods, which are password and public key

authentication. Password authentication is the easiest way to authenticate from a setup point of view, it does require the user to set public-private key authentication beforehand. When the SSH client uses password authentication, the user inputs a valid username for the system they want to access and a corresponding password, the SSH server will browse through its database of usernames and passwords and will either allow or deny authentication. The password passed between client and server is in plaintext, however, the session key created prior to authentication will encrypt any communication between two points. [10; 11.]

Public key authentication is a type of authentication which does not require any interaction between client and server, except for the initial phase of uploading the clients public key to the server. After the server has the clients public key in a file which is placed in some form of authorization file, all future public key authentication will be made automatically. The SSH client is the instigating party of public key authentication, the client sends the key pair it would like to use for authentication, the server will then check the authorization file to see if the key pair can be found. Once the key pair is found the server will encrypt a message using the public key, the client will decrypt the message and send a hash value of the decrypted message to the server. The server will compare its own generated hash of the message it sent to client, if the hash value are equal to one another the server will authenticate the user to the machine they want to access. [10; 11.]

2.3.4 Secure Copy Protocol (SCP)

SCP is a protocol and SSH client program which runs on top of the SSH protocol, used for securely copying a single file from a client machine to remote server or vice versa. SCP can also transfer multiple files recursively from a local or remote directory but cannot copy multiple directories. SCP was developed to replace the Remote Copy Protocol (RCP), just like SSH was developed to replace the RSH protocol, to allow secure data transfer between to connecting ends. [10.]

As mentioned above, SCP works on top of the SSH protocol, as Figure 2 below illustrates.

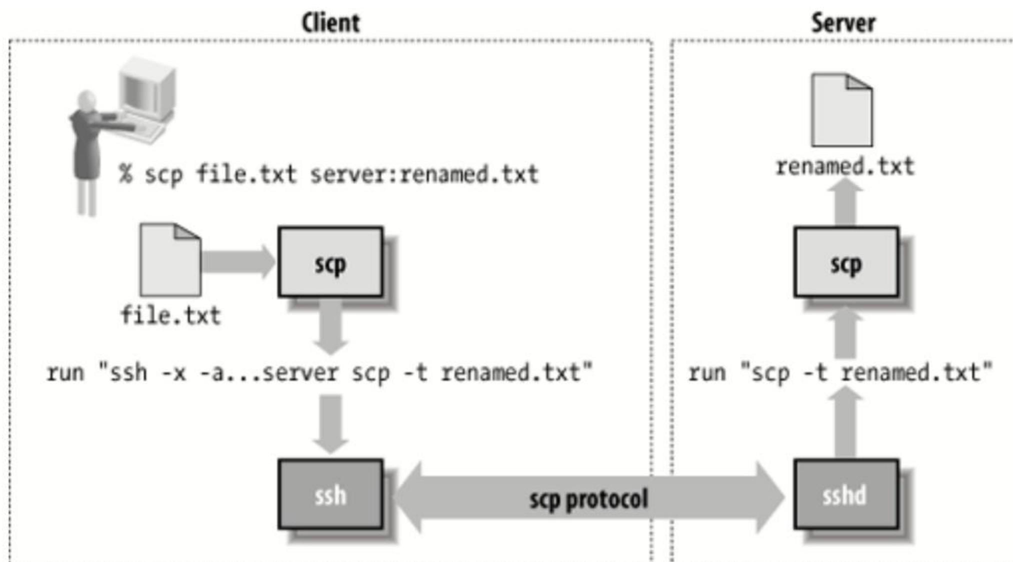


Figure 2. SCP Operation [10.]

The figure demonstrates that when a user, on the top left of the figure, uses the *scp* command, the SCP program will convert the initial command so that it combines the SSH and SCP command together. This is not visible to the user, rather this is operation is done in the background. The SSH command is required to create the initial connection to the server, ensuring the client can authenticate to the server and gain access. Once access is gained the server, represented as *sshd* in the figure, will run the SCP command and copy the file onto the machine. The simplicity by which the SCP command can be used makes it suited for single commands and is fast at transferring data because it does not require a shell to function, such as the Secure File Transfer Protocol (SFTP). Furthermore, SCP transfers data packets faster than SFTP because it does require any acknowledgement from the other end that the file has been transferred, if an SCP session is interrupted the data packet will be re-sent. [10.]

2.3.5 Disadvantages and Advantages of SSH

From a cybersecurity perspective there are a few disadvantages concerning SSH implementations, the first and most prevalent is the carelessness of authentication, especially with the use of passwords. By using passwords as an authentication method, it provides threat actors with an attack vector using brute force attacks, where attackers will continually try to break into the system using a database of the most common passwords. Moreover, if users or system administrators use default passwords to access systems it makes it easier for threat actors to crack them. There are a few mitigation options for password related issues in SSH, one can disable password authentication in the SSH server configuration and implement alternative authentication methods, such as public key authentication. Another mitigation option is to avoid default passwords, use longer and stronger passwords, and change passwords on a regular basis. [10.]

Another disadvantage of SSH implementations is their vulnerability to DOS or DDOS attacks, threat actors can flood SSH servers with authentication requests causing the server to keep unauthenticated connections open, this will eventually lead the server to reach its capacity of connections. After maximum connections have been reached new connections will not be able to connect to the server until the login grace time of the server has been reached. The login grace time determines how long a connection will stay open for authentication, after the login grace period expires the server will drop the connection. This is hard to mitigate against, however, configuring the server to have a shorter login grace period may help decrease the time the server cannot be connected to. [10.]

The advantages of implementing SSH as a means of protecting your data against threat actors are threefold. The first is that when threat actors try to eavesdrop, using a tool such as WireShark, on a connection they will not be read the data being transferred between client and server because SSH encrypts the connection and the data. The second advantage is that SSH

protects against IP spoofing where threat actors impersonate the IP address a user wants to connect to. The SSH client protects the user from connecting to the impersonated IP address by verifying the host server, by cryptographically comparing the host server public key the SSH client possesses. Finally, the third advantage of using SSH to secure s system is its ability to mitigate against Man-in-the-Middle (MITM) attacks. MITM attacks are similar to IP spoofing attacks, as they both try to coerce the SSH client into connecting to them and both imitate a SSH server. Just like with IP spoofing, the SSH client mitigates MITM attacks by verifying the public host key, furthermore, using public key authentication for the client connecting to the server ensures that the client will not connect to the nefarious server as the key would not be found. [10.]

2.3.6 SSH in the IoT

The SSH protocol is used within the IoT, especially for gaining remote access to Edge and gateway device. These devices will have an operating system running on them which require more processing power and memory. IoT systems will contain small scale computers, such as Raspberry Pis, which allow users to attach sensors and other peripherals to create IoT sensor nodes. Implementing an SSH architecture provides users and system administrators secure remote access to these devices, enabling them to update and change the behaviour of these devices. Concerning smaller constrained IoT device, such as microcontrollers, the most common network security protocol is TLS which will act as the Transport layer protocol beneath the IoT protocol, such as XMPP. [12.]

TLS and SSH are comparable to one another as they both utilize ciphers for encryption, key exchange algorithms and MACs, moreover, they are both used to encrypt and secure data in transit. Unlike SSH architecture, TLS requires an application layer protocol, such as MQTT, to handle the data before transport, whereas SSH would be able to do both from the client device to server. A benefit of using SSH over TLS is that TLS implementations can be difficult to upgrade on constrained IoT devices, as TLS libraries set security features

during software development as is integrated within the firmware of the device. SSH in contrast can function standalone without being involved in the application of the device. [12.]

2.4 Over-the-Air (OTA) Updates

The general concept of OTA updates is an update, either software or firmware, made to remote devices from a centralized location “over the air”, meaning with some form of wireless data transmission, such as Wi-Fi, 3G, 4G, Bluetooth, etc. OTA updates have been used for some time and have become a fundamental to keeping devices secure and increasing their lifespans. The most common type of OTA update experienced by people is through mobile phones or laptops. Manufacturers need to have a way of implementing new features, fix bugs, or patching security vulnerabilities without the need for customers bringing their devices to be updated manually. [13; 14.]

OTA updates in the IoT have become paramount when considering networks of IoT devices, especially due to the quickly growing domain of the IoT and the rapid pace at which manufacturers deploy their devices. As a result of rapid deployment certain features, such as functionality or security, of these devices may have been forgotten or overlooked, implementing OTA updates enables developers to fix these features. Furthermore, as development continues on the firmware or software of the microcontrollers used within the IoT, devices become more enhanced, capable, and secure. [14; 15.]

2.4.1 Benefits of OTA Updates in the IoT

There are numerous benefits to utilizing OTA updates for IoT devices, for example the ability to update a network of devices at the same time, considering a scenario where one has implemented a network of hundreds of IoT devices it would be extremely time consuming to implement updates manually. Another benefit, which was mentioned earlier, is the ability for manufacturers to implement security fixes to the devices which have been rolled

out already. Security vulnerabilities can arise at any moment after they are discovered, and it is important for these vulnerabilities to be patched quickly using OTA updates. [13.]

Implementing new or changing functionalities of IoT devices is yet another benefit of OTA updates, depending on the environment in which the device operates it would be beneficial for manufacturers to be able to change what the device can do. For example, if an IoT device has various sensors available but not all of them are needed in a system, an OTA update would allow one to disable or enable sensors to suit the requirements of the system. Finally, an important benefit of OTA updates is the ability to update devices which are found in hard to reach or remote areas. [14; 16.]

2.4.2 Concerns of OTA Updates in the IoT

There are a three main concerns of OTA updates in the IoT, firstly, is dealing with legacy IoT devices which have been deployed without foresight of implementing future fixes or device which rely on wired programming. Dealing with these types of devices will require in the field updates and will an expensive and time-consuming procedure. Secondly, there is a risk that one would make a mistake in the code of the OTA update, which would then have the undesired impact of making the device unreachable. To mitigate this concern, manufacturers will test the desired updates for device compatibility and behavioural verification on private devices before rolling out the update to the public. [14.]

Lastly, securing OTA updates is a concern that needs to be considered when implementing them, as there is always the risk that threat actors could manipulate the OTA update while it is in transit towards the device. Manufacturers need to make sure the integrity and confidentiality of the data being transferred and ensure that the devices are authenticated, some solutions include digital signatures using certificates, encryption of data, verification hash functions and message authentication codes (MAC). [16.]

This paper will focus on the last concern mentioned, by researching whether using SCP is a viable way of securing OTA updates.

2.5 IoT Devices

The following subsection of the paper will provide a brief introduction to the components which make IoT devices, followed by a technical description and choice decision of the microcontroller and sensors used in the implementation.

Basic IoT architectures comprise of three underlying layers, which is illustrated in Figure 3 below.

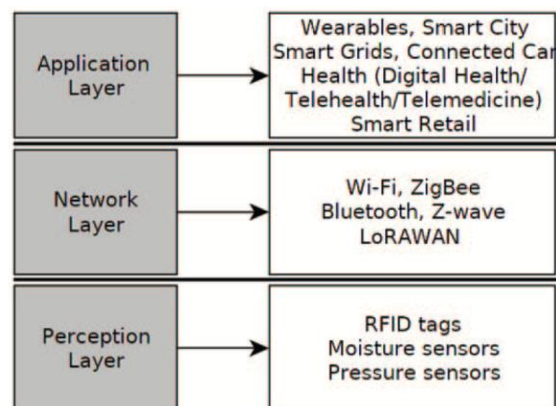


Figure 3. Basic Three-layered IoT Architecture [2.]

The perception layer is where IoT devices interact with the environment they are in, using sensors to collect physical readings or actuators which react to the environment. The network layer is what allows IoT devices to communicate to the application layer or allows applications to communicate to the IoT devices. [2.]

There are various types of IoT devices some more capable than others, it is out of the scope of this paper to describe them all, therefore it will focus on IoT devices which act as sensor nodes. Sensor nodes in the IoT comprise of hardware, generally a microcontroller, sensors, and some form of wireless communication capability. The microcontroller is a small-scale computing unit

which enables processing, power, and general-purpose input-output (GPIO) peripheral pins. By programming the microcontroller one can control the sensors attached to the GPIO pins, collecting the data they provide, then sending that data through the built-in functionality of the microcontroller, such as Bluetooth or Wi-Fi. [18.]

2.5.1 ESP-32 NodeMCU Microcontroller

The ESP-32 is an example of a widely used system on a chip (SoC) used on IoT device development boards, such as the ESP-32 NodeMCU microcontroller, which will be the device used for the implementation of this paper, illustrated in Figure 4.

Figure 4. AZ-Delivery ESP-32 NodeMCU Module [20.]

The ESP-32 SoC is versatile and has many applications, such as industrial automation, home automation, Smart buildings, Smart agriculture, healthcare applications, or even wearable devices. Furthermore, the ESP-32 SoC is used as a generic IoT sensor hub or IoT data logger, which makes it well suited for being used as a sensor node with multiple sensors in an IoT system. [19.]

There are many features of the ESP-32 SoC which can be taken advantage of, the block diagram, in Figure 5 below, illustrates the main components.

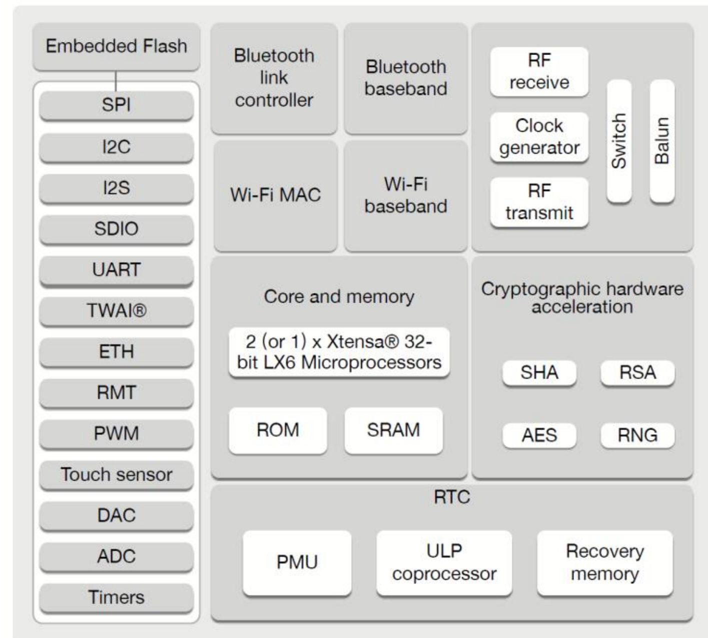


Figure 5. Block Diagram of ESP-32 SoC [19.]

It is out of the scope of this paper to investigate each component of the block diagram above. However, a focus will be made on the components important for the proposed implementation. Firstly, Wi-Fi modules of the ESP-32 provide wireless communication through the IEEE 802.11 n protocol, which functions on the 2.4 GHz frequency with up to 150 Mbps data transfer speed. Secondly, the ESP-32 SoC has a Tensilica Xtensa LX6 microprocessor, which can function as both dual-core, allowing two processors to work simultaneously on the same circuit, or single-core. [19; 20.]

Thirdly, ESP-32 SoCs internal memory components comprise of a 448 KB Read-Only-Memory (ROM), used for booting and core functionalities, a 520 KB Static-Random-Access-Memory (SRAM) for holding short term data and instructions. The third memory component is the embedded flash block, which is accessed through the Serial Peripheral Interface (SPI) of the ESP-32. The benefit of this embedded flash block, known as the Serial Peripheral Interface Flash File System (SPIFFS), is the ability for one to access the internal memory of the ESP-32 like a normal file system, allowing one to read, write, and delete files. [19.]

Lastly, the ESP-32 SoC has many peripheral options, which can be seen on the left side of the block diagram. These peripherals, which are accessed through the 34 available GPIO pins, allow the microcontroller to interact with peripheral devices, such as sensors. For example, the I2C interface uses an I2C bus to communicate between microcontroller and peripheral devices. [19.]

The ESP-32 NodeMCU in Figure 4 has a built-in ESP-32 SoC, which has all the necessary pins soldered to the SoC making it easy to include and program additional peripherals. It also uses NodeMCU which is an open-source IoT platform, this makes it easy to modify and update the firmware, and due to the open-source nature of the platform it means developers do not need to rely on proprietary software. Furthermore, it allows the use of Integrated Development Environments (IDE), such as Arduino IDE, to upload programs to desired hardware. [20.]

2.5.2 Sensors

A sensor is a device which responds to some form of physical stimulus, such as light, heat, pressure, etc., and reacts to said stimulus by transmitting an impulse [21.]. In the IoT there is a versatility of the types of sensors used and their application depends on the area they are implemented. For example, in air quality IoT systems there are sensors which measure levels of different gases in the air, in agricultural IoT systems soil moisture and humidity sensors are used, in Smart Homes temperature and luminosity sensors are utilized. [13; 17.]

This paper will provide a brief description of the two sensors used for the proposed implementation. The first sensor is the BMP180 pressure and temperature sensor, which is illustrated in Figure 6 below.

Figure 6. BMP180 Digital Pressure Sensor

The BMP180 sensor is used in weather station implementations, sport devices, vertical velocity indication or indoor and outdoor navigation systems. It has a very low impact on power and voltage consumption, which is beneficial when used with constrained IoT devices. Furthermore, it uses I2C protocol as an interface allowing easy system integration with microcontrollers. [22.]

The second sensor used within the implementation of this paper is the DHT11 sensor, shown in Figure 7 below.

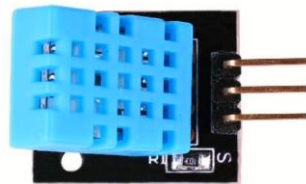


Figure 7. DHT11 Temperature and Humidity Sensor [23.]

The DHT11 is a digital temperature and humidity sensor generally used in HVAC systems, weather stations, home appliances, and is common amongst hobbyists who use microcontrollers, such as Arduino or ESP boards. It is made up of a composite sensor which uses a calibrated digital signal for temperature and humidity readings and has low power consumption. [23.]

The purpose of having two sensors for the implementation of this paper is because the BMP180 will act as the primary sensor on the sensor node. The DHT11 sensor will be the secondary sensor which will be enabled through an OTA update.

3 Implementation

The following section of this paper will describe the how SCP was implemented into an example IoT system. This IoT system includes an ESP-32 microcontroller with a BMP180 and DHT11 sensors, acting as a sensor node, and a computer running an Ubuntu operating system acting as an IoT edge device where data is collected and processed.

This section will be divided into two, the first part will describe the Arduino IDE libraries used to program the ESP-32 microcontroller, how SSH key pairs are generated and used, how SSH client is programmed and how the SSH server is set up. This is followed by the inclusion of the BMP180 into the ESP-32 microcontroller, and lastly, a description of how the sensor data was visualized. The second part of the implementation section will focus on how OTA updates were performed on this system using SCP, demonstrating how the DHT11 sensor was activated using an OTA update.

The hardware used for the implementation proposed in this paper was the NodeMCU ESP32 board, which was illustrated in Figure 4 found in the Technical Background section of the thesis. The NodeMCU ESP32 was connected to the computer running the Ubuntu operating system via a micro-USB cable, which allowed serial communication between the two devices. The serial communication between the two devices enables the Arduino IDE to upload code to the ESP-32 microcontroller.

3.1 Implementation of Secure Shell Copy

To be able to implement SCP on an ESP-32 microcontroller there were a few prerequisite tasks needed to be able to program the microcontroller. Firstly, the installation of the Arduino IDE, it is out of the scope of this paper to provide detailed instructions on how to do this, however, clear instructions are provided through the Arduino IDE website. Secondly, installing the necessary libraries to program the SCP client for ESP-32 microcontroller. This is achieved with the

following, after installing the Arduino IDE software, install the LibSSH-ESP32 library by clicking on *Tools>Manage Libraries.* This will open a prompt to search for available libraries, Figure 8 illustrates the prompt and the LibSSH-ESP32 library needed for the implementation.

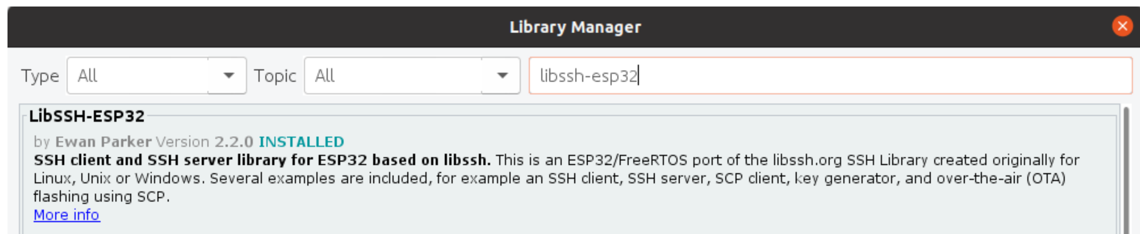


Figure 8 Arduino IDE Libraries Search Prompt with LibSSH-ESP32.

After finding the library, it is required to click *Install*, this will install the library and make it available to program the ESP-32 microcontroller with libssh functions and code. It will also be necessary to install libraries for the sensors which will be used for the ESP-32, the names of these libraries will be mentioned in subsequent sections of the paper. Installing the LibSSH-ESP32 library also provides example code which can be uploaded to the ESP-32 microcontroller, these example codes provide a base for how this library can be used. For this paper, these example codes were used with slight modifications to suit the requirements of the IoT system proposed.

3.1.1 Key Pair Generation

By implementing and utilizing SSH key pairs enables the ESP-32 microcontroller to authenticate to the SSH server. Using key pairs would be the easiest way to for the ESP-32 microcontroller gain access because using other authentication methods, such as password authentication, would not be possible due to the lack of keyboard. Key pair authentication allows the data transfer process to be automated.

To generate a public and private key for the ESP-32 microcontroller the *keygen2* example code from the LibSSH-ESP32 library was used. The *keygen2*

example code can be found by following the path in the Arduino IDE, *File>Examples>LibSSH-ESP32>keygen2*. The *keygen2* example code can generate a variety of key pairs using different algorithms, such as DSA, RSA, ECDSA, or ED25519. For the implementation of this paper a ED25519 private-public key pair was used because it provides the strongest encryption and uses the least amount of memory on the ESP-32 microcontroller. Listing 1 below provides the definition of the command to be executed to generate an ED25519 key pair for the ESP-32 microcontroller.

```
#define EX_CMD "keygen2", "--type", "ed25519", "--file",
"/spiffs/.ssh/id_ed25519"
```

Listing 1. ED25519 Key generation command.

The "*keygen2*" part of the command calls the key generation program, "*--type*" defines the type of key to be generated, which in this case is "*ed25519*". Followed by *"--file*", the format of how the key will be saved, and location for the key, "*/spiffs/.ssh/id_ed25519*".

Before uploading the code to the ESP-32 microcontroller, it was useful to modify the *keygen2* code to confirm that the key pair was generated. Furthermore, it is necessary to be able to copy the public key from the console into the OpenSSH servers *authorized_keys* file for public key authentication to work. Listing 2 opens the public key file and prints the contents of the file onto the Arduino IDE console.

```
File pubkey_file = SPIFFS.open("/.ssh/id_ed25519.pub");
if(!pubkey_file)
  {
    Serial.println("Failed to open file for reading");
    return;
   }
Serial.println("File Content:");
while(pubkey_file.available())
   {
    Serial.write(pubkey_file.read());
   }
pubkey_file.close();
```

Listing 2. Opening the public key file and printing onto the console.

When uploading code using the Arduino IDE the type of board being used must be set, for the current implementation, by selecting *Tools>Boards>DOIT ESP32 DEVKIT V1.* It is also needed to set the port through which the uploading is done, in this case selecting *Tools>Port>/dev/ttyUSB0.* After selecting the board and port uploading the code to the ESP-32 microcontroller is now possible by clicking the upload button. Figure 8 below is a screenshot of the console after the program was uploaded and executed on the ESP-32 microcontroller.

```
% No formatted SPIFFS filesystem found to mount.
% Format SPIFFS and mount now (NB. may cause data loss) [y/n]?
% Formatting...
% Mounted SPIFFS used=0 total=1378241
% Execution in progress: keygen2 --type ed25519 --file /spiffs/.ssh/id_ed25519


% Execution completed: rc=0
File Content:
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIBfAMYDv5U92eXdR8q7cfzZN+FGLEfkVwT2hO2Y2AwhQ root@esp32
```

Figure 9 Arduino Console after *keygen2* was uploaded and executed.

Figure 8 shows that the key generation was successful because the return code of the program was 0 if the code would have returned 1 this would have meant failure. The *keygen2* code will format the SPIFFS filesystem on the ESP-32 microcontroller if it does not exist on the device, this is important because it allows the generated key pair to be stored. Furthermore, formatting the SPIFFS filesystem makes it possible to create files on the device to store sensor data. Figure 8 also confirms that the public key was saved to the "*ed25519.pub"* file, the files content can be seen at the bottom of Figure 8.

The next step of setting up public key authentication is to copy the public key of the ESP-32 microcontroller into the SSH servers authorized keys file. For OpenSSH on Ubuntu, the authorized keys file is normally located in the users' home directory on the following path, "*~/.ssh/authorized_keys"*. If the file does not exist, it can be created with a text editor. After the file has been created, copy the public key from the Arduino IDE console into the *authorized_keys* file, Figure 9 below illustrates the *authorized_keys* file.

```
  GNU nano 4.8                              authorized_keys                              Modified
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIP/72PxTkEaMUWWKr9AAgEhfCtzsWOv7haIDMdZjgB6t root@esp32
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIBfAMYDv5U92eXdR8q7cfzZN+FGLEfkVwT2hO2Y2AwhQ root@esp32
ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAICdiN62tQKS2ADbpa1GQCq0YSi7SysNDG3KQDE7dWzGR alex@alex-Lenovo-Fl>
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQCihuTmCMuw+DiqTSEbAPnxujec1pYxKRWNezt65huoNiuGMyUT5YtRW79pCHi7>
```

Figure 10 Authorized keys file.

Figure 9 shows that there are several keys already within the *authorized_keys* file, the second key is the public key belonging to the ESP-32 microcontroller. When the ESP-32 microcontroller wants to authenticate to the SSH server, the server will check the authorized key file, find the public key, and verify that the device can gain access.

3.1.2 Setting Up SSH Server and SCP Client

This subsection will provide instructions on installing an OpenSSH server on an Ubuntu operating system, if it hasn't been installed already. Moreover, instructions on how to run a test on the ESP-32 microcontroller to confirm that public key authentication and secure file transfer are functional with the SCP client.

Running the following command from the terminal prompt installs the OpenSSH server on Ubuntu 20.04:

```
$ sudo apt install openssh-server
```

After the installation is complete, the SSH port must be allowed on the systems' firewall, if it is enabled, by issuing the following command:

```
$ sudo ufw allow ssh
```

It can be confirmed that the server is up and running by using the command below:

```
$ systemctl status sshd
```

Figure 10 below illustrates the desired output of the command.

```
alex@alex-Lenovo-Flex-2-Pro-15:~$ systemctl status sshd
● ssh.service - OpenBSD Secure Shell server
     Loaded: loaded (/lib/systemd/system/ssh.service; enabled; vendor preset: enabled)
     Active: active (running) since Fri 2022-04-15 21:13:27 EEST; 1 day 22h ago
       Docs: man:sshd(8)
             man:sshd_config(5)
   Main PID: 865 (sshd)
      Tasks: 1 (limit: 9303)
     Memory: 5.3M
     CGroup: /system.slice/ssh.service
             └─865 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
```

Figure 11 systemctl Command for OpenSSH Server.

The OpenSSH server is active and running, and future SSH communication between the computer and other devices is possible.

Setting up the SCP client requires using the *libssh_scp* example code found on the following Arduino IDE path, *File>Examples>LibSSH-ESP32>libssh_scp.* This initial SCP test program on the ESP-32 microcontroller will connect to the OpenSSH server running on the Ubuntu computer for the first time and store the servers' host key on the SPIFFS filesystem on the ESP-32 microcontroller. By storing the host key on the device all future communication with the OpenSSH server will be automated.

There were a few changes to the *libssh_scp* example code needed to verify that communication with the OpenSSH server and data transfer to the Ubuntu computer was successful. Firstly, providing the name of the network or Service Set Identifier (SSID) and the password for the SSID, Listing 3 shows the two variables needed for the ESP-32 microcontroller to connect to the wireless network.

```
// Set local WiFi credentials below.
const char *configSTASSID = "YourWiFiSSID";
const char *configSTAPSK = "YourWiFiPSK";
```

Listing 3. WiFi credentials for connecting to wireless network.

By configuring these variables, the ESP-32 microcontroller will gain an IP address on the local wireless network. Secondly, modifying the command to be executed, shown in Listing 4 below.

```
#define EX_CMD "libssh_scp", "/spiffs/localesp.txt",
"alex@192.168.100.36:/home/alex/testing/remotefile2.txt"
```

Listing 4. SCP command to be executed by ESP-32 microcontroller.

The command calls the SCP client with *"libssh_scp"* and tells the ESP-32 microcontroller to copy the local file, "*localesp.txt"*, found on the ESP-32 SPIFFS filesystem, to a remote file named "*remotefile2.txt"* located on the Ubuntu computer running the OpenSSH server.

Thirdly, creation of the local file, "*localesp.txt",* on the ESP-32 microcontroller before sending the file to the remote computer. Listing 4 provides the code that creates a file local file on the ESP-32 microcontroller.

```
File data_t = SPIFFS.open("/localesp.txt", FILE_WRITE);
if (!data_t)
{
  Serial.println("There was an error opening the file for writing");
  return;
}
if (data_t.print("testing testing \n"))
{
  Serial.println("File was written");
} else
{
  Serial.println("File write failed");
}
data_t.close();
```

Listing 5. Creating local file on ESP-32 microcontroller

The code demonstrated in Listing 5 creates a file named "*localesp.txt"* and writes "*testing testing"* into the file. The file creation occurs in a function named "controlTask", this is the main function of the program where the SCP client command is executed from. The final modification before uploading the program to the ESP-32 microcontroller is to demonstrate to the user that the host key of the server was saved into the SPIFFS filesystem. Using the same algorithm demonstrated in Listing 2, the file "*.ssh/known_hosts"* on the ESP-32 microcontrollers' filesystem will be opened and printed on the Arduino IDE

console. The “*known_hosts”* file is the location where SSH server host keys are stored, allowing the device to confirm it has connected to the correct server found on that IP address.

The stage, after modifying the example code, was to upload the code to the ESP-32 microcontroller. Figure 12 provides the Arduino IDE console output after the program was uploaded.

```
% Mounted SPIFFS used=1255 total=1378241
File was written
% WiFi enabled with SSID=
% IPv6 Address: fe80:0000:0000:0000:cac9:a3ff:fed2:9dec
% IPv4 Address: 192.168.100.44
% Timeout waiting for IP address
% Execution in progress: libssh_scp /spiffs/localesp.txt alex@192.168.100.36:/home/alex/testing/remotefile2.txt

The server is unknown. Do you trust the host key (yes/no)?
SHA256:di/rgT7Fv69ulbOSpPZcI1OHEW4mvG09/kkMCwC0UAE
This new key will be written on disk for further usage. do you agree ?
wrote 15 bytes

% Execution completed: rc=0
```

Figure 12 First Execution of libssh_scp Console Output.

The first execution of the program requires interaction from the user, Figure 12 illustrates that user will need to decide if the server can be trusted and if the server key should be saved to the ESP-32 microcontroller. By answering “yes” to both questions it will not be necessary to interact with the device anymore, making the data transfer process fully automated. Figure 13 below demonstrates that no future interaction is needed.

```
% Mounted SPIFFS used=2259 total=1378241
File was written
File Content:
192.168.100.36 ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIEn/LR4X4VvINJoM8Ph1GcpDEa5qpPtHUb6rrCTkEzj0
% WiFi enabled with SSID=Koti_u75P
% IPv4 Address: 192.168.100.44
% IPv6 Address: fe80:0000:0000:0000:cac9:a3ff:fed2:9dec
% Timeout waiting for IP address
% Execution in progress: libssh_scp /spiffs/localesp.txt alex@192.168.100.36:/home/alex/testing/remotefile2.txt

wrote 17 bytes

% Execution completed: rc=0
```

Figure 13 Second Execution of libssh_scp Program.

The second execution of the program also allows the user to see that the host key was written to the “*known_hosts”* file on filesystem of the ESP-32

microcontroller, as it can be seen on the fourth line of the output. Figure 12 and Figure 13 also shows that the ESP-32 microcontroller was able to connect to the wireless network and gained both IPv4 and IPv6 addresses. By being connected to the network the SCP client command was able to execute and copying the local file on the ESP-32 microcontroller to the remote file was successful. Figure 14 below is a screen shot of the transferred file on the remote Ubuntu computer.

```
alex@alex-Lenovo-Flex-2-Pro-15:~/testing$ ls
remotefile2.txt
alex@alex-Lenovo-Flex-2-Pro-15:~/testing$ cat remotefile2.txt
testing testing
```

Figure 14 Transferred File to Ubuntu Computer

By checking that the transferred file can be found on the remote computer and that the contents of the file match it can be confirmed that the SCP client data transfer was successful. Testing the *libssh_scp* example code was a useful method of allowing the ESP-32 microcontroller to become aware of the OpenSSH server by saving the servers' host key, making future communication possible without the need for device interaction. The next subsection of the paper provides a description of how the sensors were included into the ESP-32 microcontroller.

3.1.3 Including BMP Sensor: Collecting and Sending Data via SCP

The following stage of the implementation incorporates the BMP180 and DHT11 sensors with the ESP-32 microcontroller. The first step was to wire the sensors into the GPIO pins of the microcontroller, which is illustrated in the wire diagram in Figure 15.

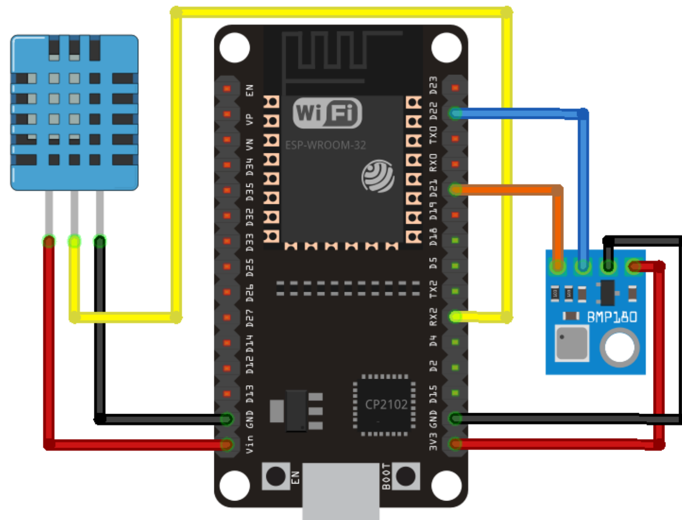


Figure 15 Wire Diagram of ESP-32, BMP180 Sensor and DHT11 Sensor.

Both sensors were included in this stage of the implementation, however, the DHT11 sensor on the left of Figure 15 would be enabled during the OTA update implementation.

Interaction between the ESP-32 microcontroller and the BMP180 sensor was possible through the installation of the Adafruit BMP085 Library, shown in Figure 16.

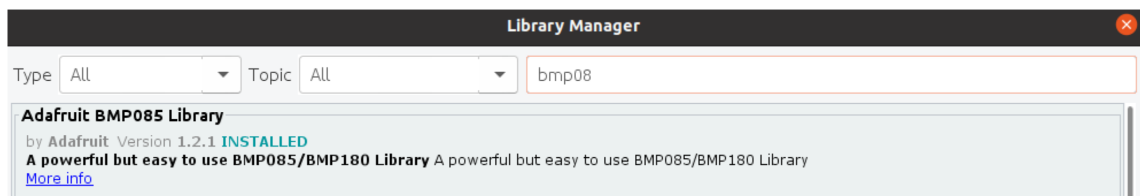


Figure 16 Adafruit BMP085 Library Installation.

It was also necessary to install a generalized sensor library named Adafruit Unified Sensor, illustrated in Figure 17.

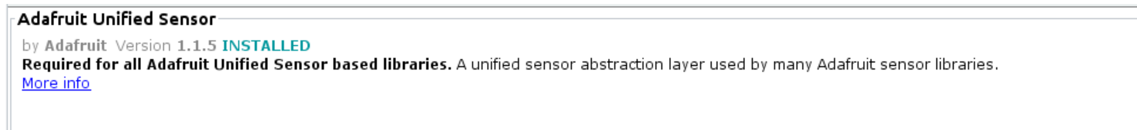


Figure 17 Adafruit Unified Sensor Library Installation.

After the installation of the two libraries above in the Arduino IDE, it was possible to include the necessary header files for the ESP-32 microcontroller to communicate with the BMP180. The final library which was installed was the ESPDateTime library shown in Figure 18.

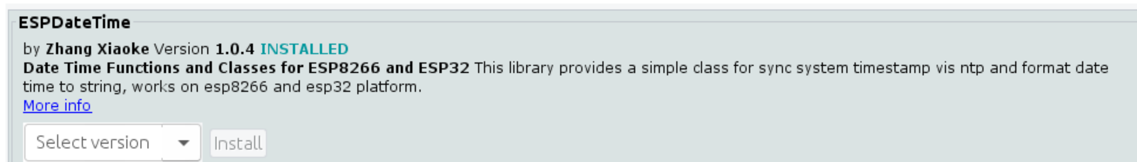


Figure 18 ESPDateTime Library Installation

The ESPDateTime library uses the Network Time Protocol (NTP) to synchronize the date and time of the ESP-32 microcontroller with a network server which provides the time. It was important to include this library as it provided accurate timestamps for when the sensor data was taken.

The next step of incorporating the BMP180 sensor was to include a new task for the ESP-32 microcontroller using the same code base from the previous section. As it was not necessary to create an example file or view the content of the "*known_hosts"* file, these sections were removed. Listing 6 below shows which header files and object creation were needed to interact with the BMP180 sensor.

```
#include <Wire.h>
#include <Adafruit_BMP085.h>
#include "ESPDateTime.h"

Adafruit_BMP085 bmp;
```

Listing 6. Header files and bmp Object creation.

The first two header files in Listing 6 enable the ESP-32 microcontroller to use the functions to communicate with the BMP180 sensor and the third enable functions to access the NTP server. The fourth line in Listing 6 declares the BMP180 object "*bmp*", this part of the program tells the ESP-32 microcontroller to read the sensor data through the "*bmp*" object. These lines were added before the setup function of the code, the setup function acts as the main function of the program.

Appendix 1 of the paper provides the sensor task function which was added to the example *libssh_scp* code used in the SCP testing phase of the implementation. The function "*Task_Sensor()*" initializes the BMP180 sensor and begins reading the data coming from the sensor, it will also initialize the NTP time synchronization. The function will create a character array to store the data to be saved into Comma Separated Value (CSV) file to be transferred to the remote SSH server. Listing 7 shows the function used to store the timestamp and sensor data value.

```
sprintf(tempall, "%s, %f \n", DateTime.format(DateFormatter::SIMPLE).c_str(),
temp);
```

Listing 7. sprintf Function.

The *sprintf* function will write the timestamp string and the BMP180 sensors temperature floating-point number value, "*temp*", into the character array defined as "*tempall*". The sensor task function will then create a file named "*sensor_data.csv*" on the ESP-32 microcontrollers filesystem and store the character array within the file. For this function to be used it must be defined in the setup function of the code after the SCP client control task function. Listing 8 shows the two tasks which will be run on the ESP-32 microcontrollers' processor.

```
xTaskCreatePinnedToCore(controlTask, "ctl", configSTACK, NULL, 3, NULL, 1);
xTaskCreatePinnedToCore(Task_Sensor, "Task_Sensor", 2048, NULL, 2, NULL, 1);
```

Listing 8. ESP-32 processor core tasks.

The SCP client command control task and the sensor reading and collecting task, defined in the setup function, are both run from the ESP-32 microcontrollers' first core processor using the "*xTaskCreatePinnedToCore"* function seen in Listing 8. The fifth variable taken by the function is priority level and the seventh variable defines which core processor will be used. The sensor task takes higher precedence as it is more important to collect the sensor data than send the data to the remote computer.

The next step of the implementation was to modify the "*opts*" function in the code. The "*opts"* function iterates through the arguments given to the *libssh_scp* command, such as the name of the file to be copied from the ESP-32 microcontroller to remote computer. The modification is shown in Listing 9 below.

```
sources[i] = NULL;
destination = argv[optind];
optind = 0;
return 0;
```

Listing 9. Added definition for *optind* variable.

Adding the redefinition of the variable "optind" to zero, found at the end of the "*opts"* function, will set the libssh_scp command arguments to the beginning, otherwise when the control task loops the iteration will begin at an argument variable which does not exist.

The final changes to the code were to set the control task and sensor task to loop every ten minutes, this was achieved by adding the line shown in Listing 10 at the end of the while loops found in each task.

```
vTaskDelay(600000);
```

Listing 10. Task delay function.

The "*vTaskDelay"* function will delay the task it resides in, for the purpose of this implementation it was set to ten minutes. The last change was to redefine the

SCP client command to be executed by the program, Listing 10 illustrates the changes made.

```
#define EX_CMD "libssh_scp", "/spiffs/sensor_data.csv",
"alex@192.168.100.36:/home/alex/testing/data.csv"
```

Listing 11. SCP command to be executed.

The command to be executed call the "*libssh_scp"* program to copy the newly created "*sensor_data.csv"* file from the ESP-32 microcontrollers' filesystem to a file called "*data.csv"* found on the remote computer running the SSH server. Once these changes have been made to the code uploading the program to the ESP-32 microcontroller was possible. Figure 19 below illustrates two iterations of transferring the sensor data to the remote computer.

```
% Mounted SPIFFS used=3263 total=1378241
% WiFi enabled with SSID=
% IPv4 Address: 192.168.100.44
% IPv6 Address: fe80:0000:0000:0000:cac9:a3ff:fed2:9dec
File was written
% Timeout waiting for IP address
% Execution in progress: libssh_scp /spiffs/sensor_data.csv alex@192.168.100.36:/home/alex/testing/data.csv

wrote 32 bytes

% Execution completed: rc=0
File was written
% Execution in progress: libssh_scp /spiffs/sensor_data.csv alex@192.168.100.36:/home/alex/testing/data.csv

wrote 32 bytes
```

Figure 19 Arduino Console Output of SCP Client Command.

It can be seen from Figure 19 that the first iteration of the program will ensure that the SPIFFS filesystem has been mounted on the ESP-32 microcontroller, and that it has an IPv4 and IPv6 address. It will also write the sensor data to the "*sensor_data.csv"* file, followed by the execution of the SCP client command. The second and future iterations will rewrite the file and replace the file on the remote computer. To ensure that the "*data.csv"* file was created on the remote computer it is healthy to verify that it was created, Figure 20 confirms that file was created.

```
alex@alex-Lenovo-Flex-2-Pro-15:~/testing$ cat data.csv
2022-04-19 15:44:50, 24.799999
```

Figure 20 CSV File on Remote Computer.

Figure 20 demonstrates that the SCP client command was successful from the ESP-32 microcontroller to the remote computer. Figure 20 also confirms that the timestamp of when the sensor data was taken, and the temperature value was written to the file.

3.1.4 Visualization of Sensor Data

The final stage of completing this implementations' IoT system was to visualize the sensor data. To visualize the data a Python script was utilized, the script had to achieve a few things, firstly, to copy the data from the file which was transferred from the ESP-32 microcontroller to a file which would hold all the collected data. Listing 12 illustrates the lines of code in the Python script which parsed the received data into a new file.

```
fout = open("graph_data.csv", "a")
for line in open("data.csv"):
    fout.write(line)
fout.close()
```

Listing 12. Python script moving received data into new file.

The "*data.csv*" file in Listing 12 is the file which was transferred from the ESP-32 microcontroller to the remote computer using SCP. The CSV file, "*graph_data.csv*", is the file which holds all the collected sensor data. The Python script will parse the line within the "*data.csv*" file and append it to the "*graph_data.csv*" file. Figure 21 below illustrates the outcome of the Python script file addition.

```
alex@alex-Lenovo-Flex-2-Pro-15:~/testing$ cat graph_data.csv
timestamp,temperature,humidity
2022-04-19 15:34:50, 24.799999
2022-04-19 15:44:50, 24.799999
2022-04-19 15:54:50, 24.700001
2022-04-19 16:04:50, 24.600000
2022-04-19 16:14:50, 24.500000
```

Figure 21 CSV File Containing Sensor Data.

CSV file format was chosen for the implementation because it is a simple file format which can be easily parsed and converted into graphs using Python scripting. Figure 21 shows that "*graph_data.csv*" contains three titles, "timestamp, temperature, humidity", these titles represent the data points to be plotted on the graph. However, humidity data was not collected yet and will be used after the OTA update in the next subsection of the paper.

A second Python script was needed to ensure that the data from the received file would be added to the larger data file found on the remote computer. This was achieved using a Python script which acted as a "watchdog" over the "*data.csv"* file. Listing 13 provides a snippet of the Python script watching over the data file.

```
class Handler(FileSystemEventHandler):
    def on_modified(self, event):
        if event.src_path == "./data.csv":
            print ("changed")
            subprocess.call(['python3', 'csv_viz2.py'])
```

Listing 13. Watchdog Python script.

The script uses the "Handler" class to wait for an event to occur on the file system, and if statement is triggered when a change occurs on the "*data.csv"* file. Once a change occurs on the file the script will trigger a subprocess to run the "*csv_viz2.py"* file, which is the Python script that merges the received data file to the larger CSV file holding all of the received sensor data.

The first Python script will also plot the data points into a graph and save the graph as a PNG file which was inserted into a simple HTML file to be shown on

a website. Figure 22 illustrates the final visualized data on a website running on the remote Ubuntu computer.

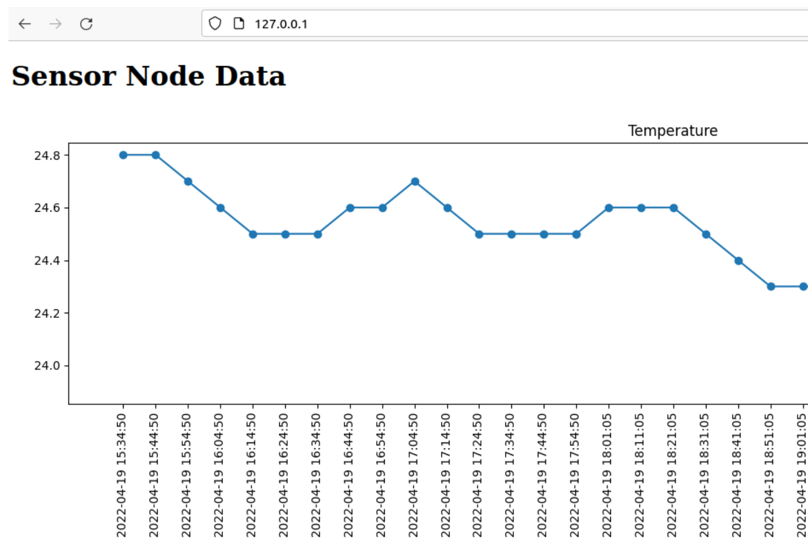


Figure 22 Temperature Data Visualization on Web Browser.

Figure 22 visualizes the sensors temperature data values, with x-axis representing the timestamps of when the sensor value was taken and y-axis representing the temperature values.

3.2 Implementation of OTA Updates

Implementing OTA updates for the ESP-32 microcontroller using the same code used for SCP file transfers required using the “*otaTask*” function defined in the *FirmwareOTAClientSCP* example code file found in the Arduino IDE. After locating the function, it was copied into the SCP file transfer code. The “*otaTask”* function connects to the OTA server, which was the same computer running the SSH server, and locates the compiled binary of the program which would be used for the update in a defined directory. Listing 14 provides the server, user, and file location definitions.

```
const char *configOTAServer = "your.scpserver.local";
const char *configOTAUser = "username";
const char *configOTAPath = "/path/to/firmware.ino.bin/file";
```

Listing 14. OTA Definitions.

To be able to use the “otaTask” function including the header file in Listing 15 was needed.

```
#include "esp_ota_ops.h"
```

Listing 15. OTA header file inclusion.

For the OTA update to occur on a regular basis, a while loop containing the contents of the functions and a task delay was added within the *“otaTask”* function. The next step was to define the task in the setup function of the code, shown in Listing 16.

```
delay(20000);
xTaskCreatePinnedToCore(otaTask, "ota", configSTACKota, NULL, 4, NULL, 1);
```

Listing 16. Task definition in setup function.

A delay was also needed before starting the OTA task because it requires the initial wireless network to be configured before it can begin checking for an update. After these changes were made it was possible to upload the program to the ESP-32 microcontroller. Figure 23 illustrates the output of the uploaded program on the Arduino IDE console.

```
% Mounted SPIFFS used=3263 total=1378241
% WiFi enabled with SSID=
% IPv4 Address: 192.168.100.44
% IPv6 Address: fe80:0000:0000:0000:cac9:a3ff:fed2:9dec
File was written
% Timeout waiting for IP address
% Execution in progress: libssh_scp /spiffs/sensor_data.csv alex@192.168.100.

wrote 32 bytes

% Execution completed: rc=0
ota% Task started
ota% Boot Partition type=0 subtype=16 addr=65536 size=1310720 label=app0
ota% Run Partition type=0 subtype=16 addr=65536 size=1310720 label=app0
ota% OTA Partition type=0 subtype=17 addr=1376256 size=1310720 label=app1
ota% ssh_scp_pull_request failed rc=5 SCP: Warning: scp: /home/alex/esp_upgra
ota% Skipped
```

Figure 23 Arduino Console Output After OTA Task Inclusion.

The last six lines of the output represent what occurred during the OTA update, the update was not successful because the binary file for the update was not located. Creating a binary file for the OTA update requires selecting *Sketch>Export compiled binary* in the Arduino IDE and copying the binary file to OTA path defined in the code. For testing purposes, the first compiled binary included following line shown in Listing 17.

```
Serial.println("OTA UPDATE TEST 1");
```

Listing 17. Function printing line to Arduino IDE console.

The line of code in Listing 17 will print out “OTA UPDATE TEST 1” on to the console. Figure 24 provides the output of the Arduino IDE console after the OTA update was completed.

```
% Mounted SPIFFS used=3263 total=1378241
% WiFi enabled with SSID=
File was written
OTA UPDATE TEST 1
```

Figure 24 Arduino Console After OTA Update Completion.

It can be seen from Figure 24 that the update was successful as the line “OTA UPDATE TEST 1” is present. After making sure the initial update tasks were successful, the next step of the implementation was to enable the DHT11 sensor on the ESP-32 microcontroller.

Firstly, for the ESP-32 microcontroller to communicate with the DHT11 sensor the DHT11 sensor library, pictured in Figure 25, had to be installed.

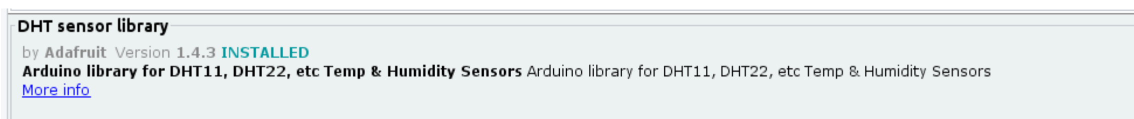


Figure 25 DHT Sensor Library Installation.

Secondly, adding the lines, seen in Listing 18 below, included the header file needed for the DHT11 sensor, the definition of the ESP-32 microcontrollers' GPIO pin the DHT11 sensor was attached to, and creation of the DHT11 object to be used in the code.

```
#include <DHT.h>
#define DHT11PIN 16
DHT dht(DHT11PIN, DHT11);
```

Listing 18. Including DHT11 sensor header file, GPIO pin definition, DHT object creation.

Thirdly, using the same sensor task containing the BMP180 sensor the DHT11 sensors' initializing and value reading function were added. Furthermore, writing the collected sensors values to the "*sensor_data.csv"* file was included into the "*Task_Sensor"* function. Listing 19 provides the two lines added and one modified line of code.

```
dht.begin();
float hum = dht.readHumidity();

sprintf(tempall, "%s, %f, %f \n", DateTime.format(DateFormatter::SIM-
PLE).c_str(), temp, hum);
```

Listing 19. DHT11 sensor incorporation into sensor task function.

The first line in Listing 19 initializes the DHT11 sensor to be used, the second line defines a floating-point number variable to store the DHT11 sensors' humidity reading value. The last line is modified to include the DHT11 sensors' humidity value into the character array used before.

The last step of implementing the OTA update for enabling the DHT11 sensor was to generate the compiled binary file and copy the file to the defined OTA path. The ESP-32 microcontroller will update itself using an OTA update after the delay of OTA task has been reached. Checking the "*data.csv"* file received from the ESP-32 microcontroller confirmed that the update was successful. Figure 26 shows the content of the "*data.csv"* file.

```
alex@alex-Lenovo-Flex-2-Pro-15:~/testing$ cat data.csv
2022-04-21 17:02:41, 24.500000, 22.000000
```

Figure 26 Content of Received File.

Figure 26 demonstrates that the new humidity value from the DHT11 sensor was successfully added into the CSV file after the temperature value from the BMP180 sensor. To complete the IoT system in this implementation, the DHT11 sensor humidity values were plotted on the same graph using the same Python script from the previous section. Figure 27 illustrates the sensor nodes temperature and humidity values plotted on a graph shown on a web browser.

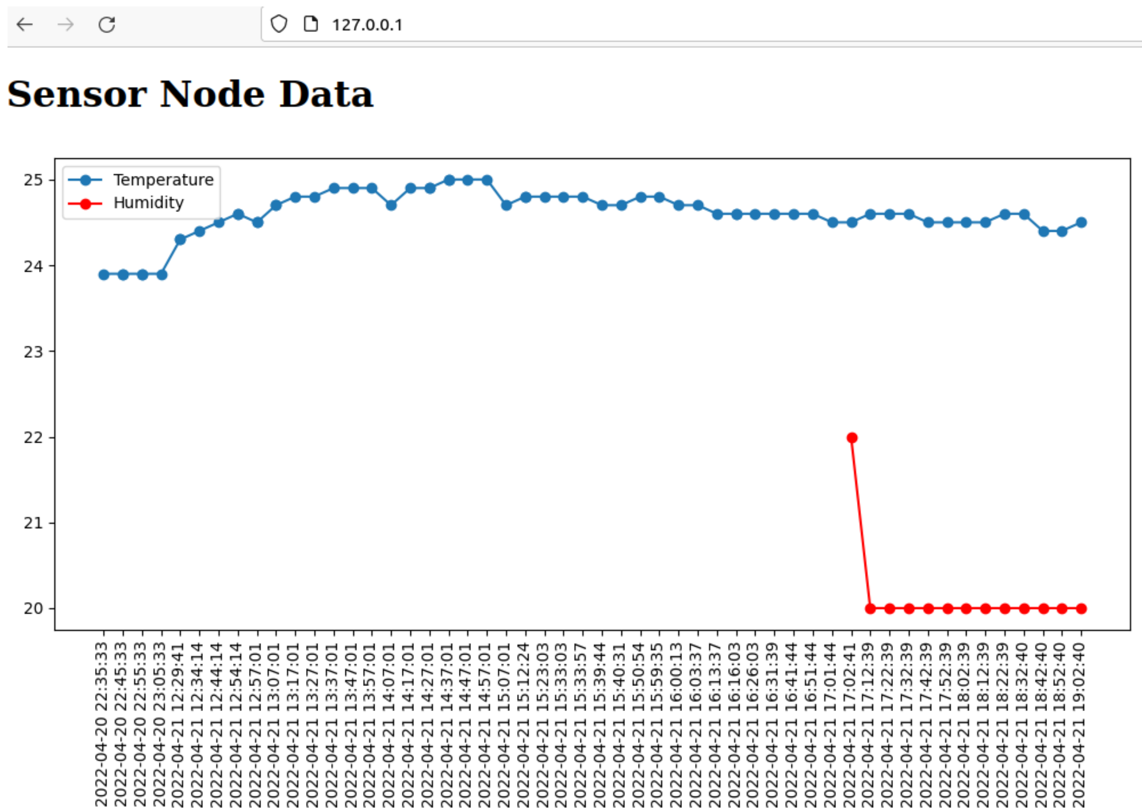


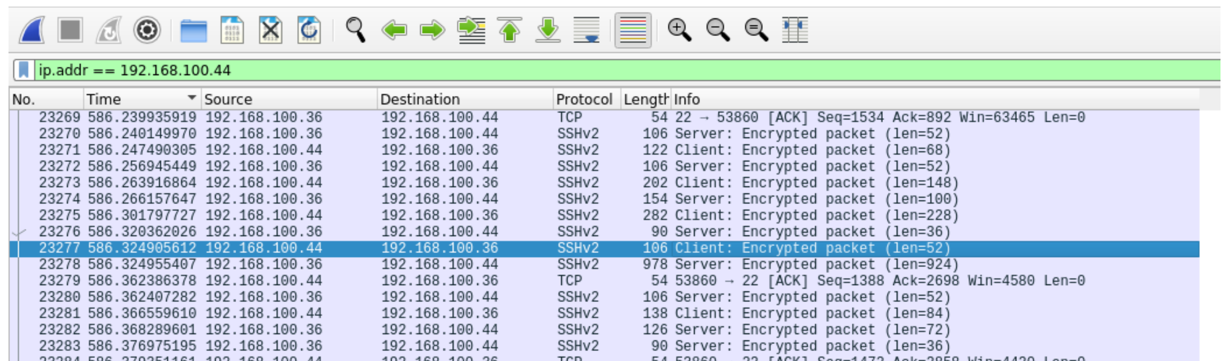
Figure 27 ESP-32 Sensor Node Web Browser Graph.

The graph in Figure 27 shows the temperature values from the BMP180 sensor represented by the blue line and humidity values from the DHT11 sensor in red. The humidity values began after the OTA update was completed.

4 Results

The following section will review the results of the implementation from a security perspective, by analysing whether the data packets transferred between two end devices was encrypted and secured.

The goal of the proposed implementation of securing the sensor data in transit and performing secure OTA updates using SCP was achieved. WireShark, a network packet analyser software, was used to verify that the data was encrypted during transit. Figure 28 provides a snapshot of the WireShark analysing the network traffic coming from the ESP-32 microcontrollers' IPv4 address, 192.168.100.44.



No.	Time	Source	Destination	Protocol	Length	Info
23269	586.239935919	192.168.100.36	192.168.100.44	TCP	54	22 → 53860 [ACK] Seq=1534 Ack=892 Win=63465 Len=0
23270	586.240149970	192.168.100.36	192.168.100.44	SSHv2	106	Server: Encrypted packet (len=52)
23271	586.247490305	192.168.100.44	192.168.100.36	SSHv2	122	Client: Encrypted packet (len=68)
23272	586.256945449	192.168.100.36	192.168.100.44	SSHv2	106	Server: Encrypted packet (len=52)
23273	586.263916864	192.168.100.44	192.168.100.36	SSHv2	202	Client: Encrypted packet (len=148)
23274	586.266157647	192.168.100.36	192.168.100.44	SSHv2	154	Server: Encrypted packet (len=100)
23275	586.301797727	192.168.100.44	192.168.100.36	SSHv2	282	Client: Encrypted packet (len=228)
23276	586.320362026	192.168.100.36	192.168.100.44	SSHv2	90	Server: Encrypted packet (len=36)
23277	586.324905612	192.168.100.44	192.168.100.36	SSHv2	106	Client: Encrypted packet (len=52)
23278	586.324955407	192.168.100.36	192.168.100.44	SSHv2	978	Server: Encrypted packet (len=924)
23279	586.362386378	192.168.100.44	192.168.100.36	TCP	54	53860 → 22 [ACK] Seq=1388 Ack=2698 Win=4580 Len=0
23280	586.362407282	192.168.100.36	192.168.100.44	SSHv2	106	Server: Encrypted packet (len=52)
23281	586.366559610	192.168.100.44	192.168.100.36	SSHv2	138	Client: Encrypted packet (len=84)
23282	586.368289601	192.168.100.36	192.168.100.44	SSHv2	126	Server: Encrypted packet (len=72)
23283	586.376975195	192.168.100.36	192.168.100.44	SSHv2	90	Server: Encrypted packet (len=36)

Figure 28 WireShark Packet Capture from ESP-32 Microcontroller

The snapshot pictured in Figure 28 provides a packet analysis of the data communication between the OpenSSH server running on the remote computer represented with the IPv4 address 192.168.100.36 and the ESP-32 microcontroller. This is a capture of copying the sensor data CSV file from the ESP-32 microcontroller to the remote computer, the protocol used for data communication was SSHv2. This signifies that the implementation of the SCP client on the ESP-32 microcontroller was successful in utilizing the SSH protocol to encrypt data between two end points, which is illustrated in Figure 29.

```
▾ SSH Protocol
  ▾ SSH Version 2 (encryption:aes256-gcm@openssh.com mac:<implicit> compression:none)
      Packet Length: 208
      Encrypted Packet: af8f03db08e3d8cecb7efb0ba824fd1401e74814c703b546…
      MAC: 64bf5861bce15dc98c9288792580aa75
    [Direction: client-to-server]

0000  d0 7e 35 96 71 7f c8 c9  a3 d2 9d ec 08 00 45 00   ·~5·q···  ······E·
0010  01 0c 01 12 00 00 ff 06  70 38 c0 a8 64 2c c0 a8   ········  p8··d,··
0020  64 24 d2 64 00 16 57 53  42 10 79 92 c5 8a 50 18   d$·d··WS  B·y···P·
0030  15 a4 27 0c 00 00 00 00  00 d0 af 8f 03 db 08 e3   ··'·····  ········
0040  d8 ce cb 7e fb 0b a8 24  fd 14 01 e7 48 14 c7 03   ···~···$  ····H···
0050  b5 46 d9 a8 2f 01 ac e4  4f d4 0c c3 f2 03 d1 d7   ·F··/···  O·······
0060  a0 59 30 6a f5 b5 a9 df  56 7e 86 68 f7 5d 73 93   ·Y0j····  V~·h·]s·
0070  0f 58 da a4 8a e7 94 b1  d3 c6 20 21 d5 95 55 fe   ·X······  ·· !··U·
0080  34 e1 d4 54 00 1c 0c 85  81 ee aa 3b 8e 49 4b ae   4··T····  ···;·IK·
0090  45 b9 f2 f5 7c c8 14 6a  97 0c db 07 ed 2b d7 dd   E···|··j  ·····+··
00a0  57 d9 81 ce 1d 99 ec 4e  4b 49 b3 d9 f8 04 52 03   W······N  KI····R·
00b0  37 8e e2 6f d1 77 a8 15  be 47 3d 01 96 85 c9 63   7··o·w··  ·G=····c
00c0  f3 84 cf ee ae c7 cc f2  dd 22 cf 9f bf cc e8 55   ········  ·"·····U
00d0  6b 42 94 e6 ae e4 c3 a0  8a 64 7b b6 6d 73 0a 21   kB······  ·d{·ms·!
00e0  3d 3e 31 26 1a ff 84 24  ee 4f 8e de 82 f7 c7 da   =>1&···$  ·O······
00f0  68 3d 6a 62 4e 76 49 80  2e 03 df 70 26 59 64 f0   h=jbNvI·  .··p&Yd·
0100  eb 0a ad d2 6a 01 9e ee  0a 03 64 bf 58 61 bc e1   ····j···  ··d·Xa··
0110  5d c9 8c 92 88 79 25 80  aa 75                     ]····y%·  ·u
```

Figure 29 WireShark View of Packet Content

Figure 29 illustrates an analysis of one packet moving from the ESP-32 microcontroller, the client, to the remote computer, the server. The data within the packet has been encrypted because the data has been converted into random alpha numeric characters seen on the bottom right of Figure 29.

Figures 28 and 29 were an example of one communication between the ESP-32 microcontroller and remote computer running the SSH server. However, all communication using the SCP client programmed into the ESP-32 microcontroller was encrypted, including the OTA updates because they used the SCP client program to copy the updated firmware code from the remote computer to ESP-32 microcontroller.

5 Discussion

The following section will discuss the limitations of the implementation and provide an assessment for possible future development.

5.1 Limitations

There were a few limitations concerning the implementation of the SSH protocol and SCP client programs into an IoT system. Firstly, the proposed implementation focused on one type of IoT device, the ESP-32 microcontroller. Focusing on an individual device meant that it was not possible to verify that an SSH implementation would be able to work on the many available devices found in the IoT device market. The versatility of the devices implies a difference in computing and processing power, this means some devices may not be able to handle the use of the SSH protocol. Furthermore, there are IoT devices which are constrained due to their power consumption, an aspect which was not tested during the implementation phase of this paper.

Secondly, relating to the use of an individual device, it was not possible to foresee how the proposed implementation would work in a production environment with hundreds or thousands of IoT devices. Increasing the number of devices would require the consideration of how to implement the system on a larger scale. For example, answering the question of how to implement individual SSH key pairs for each device or applying specific OTA updates to devices found in different environments.

Lastly, a method to determine if the ESP-32 microcontroller had received and applied an OTA update was not researched during the implementation phase of the paper. With the current implementation there is no way for the user to know if the IoT device was updated, the only mechanism to verify the update was through the received CSV data file or the change in the visualization of the data found in the web browser.

5.2 Future Development

There are a few factors which should be considered for the future development of the implementation suggested in this paper. Testing and applying the implementation on a wider range of IoT devices would aid in developing a generalised method of implementing the SSH protocol. Evaluating the implementation on other devices would also help determine which devices would be able to utilize the proposed SSH system. Moreover, analysing the power consumption of when the SCP client program was used and comparing the power consumption of other IoT protocols would help influence the selection of which IoT protocol should be used.

Another consideration for future development should be a comparison of using SCP as a means of transferring data and other available IoT protocols. Comparing variables, such as the speed of transfer, size of the data packets, and resource consumption would be beneficial in creating a benchmark to determine which protocol would be best suited for a proposed IoT system. For example, being aware of the size of the transferred data packets would provide IoT system architects with knowledge to decide if a constrained networks bandwidth could handle the data being transferred.

Furthermore, future research into the development of a mechanism to verify OTA updates of a device would aid in removing the doubt of whether a device was updated. Moreover, this verification mechanism could also include a security aspect, by ensuring that the OTA update was implemented from the correct server and not from a threat actor using a MITM attack who imitates the server. The security verification could include a form of hash verification to ensure the OTA update came from the correct server.

6 Conclusion

The use of the SSH protocol in the IoT is primarily focused on using it as a secure remote access tool rather than a data transport protocol. The purpose of this paper was to research the viability of implementing the SSH protocol through an SCP client program as a means of securely transporting data in an IoT system. Firstly, by providing an analysis of the current cybersecurity threats faced by IoT systems and IoT devices and a breakdown of the weaknesses and strengths of popular IoT protocols currently used. Furthermore, an in-depth analysis of the SSH protocol and the importance of OTA updates of IoT devices provided insight into the possibility of their utilization within an IoT system.

Secondly, this paper provided the means of implementing an SSH architecture into an IoT system. The system proposed consisted of a sensor node, comprised of ESP-32 microcontroller, BMP180 sensor, and DHT11 sensor, and an IoT edge device running an SSH server. The edge device was a computer running an Ubuntu operating system and was the device where the sensor nodes' data was processed and visualized in a web browser running on the computer. Moreover, the implementation provided evidence of the viability of applying an SSH architecture to an IoT system to securely transfer sensor data and implement secure OTA updates to enable features on IoT devices.

Thirdly, the results section of the paper demonstrated that by using the WireShark network packet analyser software, the data being transferred between the sensor node and edge device were encrypted and secure from eavesdroppers. Lastly, a discussion of the proposed IoT system provided the limitations faced during the implementation phase of the paper. In addition, future development considerations were outlined to further improve the application of SSH architectures to secure data in transit within IoT systems.

References

1 Harper A. Gray hat hacking: the ethical hacker's handbook. Fifth edition. New York: McGraw-Hill Education; 2018.

2 Karie NM, Sahri NM, Haskell-Dowland P. IoT Threat Detection Advances, Challenges and Future Directions. In: 2020 Workshop on Emerging Technologies for Security in IoT (ETSecIoT) [online]. Sydney, Australia: IEEE; 2020. pp. 22–9.
URL: https://ieeexplore.ieee.org/document/9097762/. Accessed 4 March 2022.

3 P. Radanliev, D. De Roure, S. Cannady, R. M. Montalvo, R. Nicolescu and M. Huth. Economic impact of IoT cyber risk - Analysing past and present to predict the future developments in IoT risk analysis and IoT cyber insurance. In: *Living in the Internet of Things: Cybersecurity of the IoT – 2018* [online]. London, United Kingdom: IET; 2018. pp. 1-9.
URL: https://ieeexplore.ieee.org/document/8379690/. Accessed 6 March 2022.

4 Bandekar A, Javaid AY. Cyber-attack Mitigation and Impact Analysis for Low-power IoT Devices. In: 2017 IEEE 7th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER) [online]. Honolulu, HI: IEEE; 2017. pp. 1631–6.
URL: https://ieeexplore.ieee.org/document/8446380/. Accessed 4 March 2022.

5 Gurunath R, Agarwal M, Nandi A, Samanta D. An Overview: Security Issue in IoT Network. In: 2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) [online]. Palladam, India: IEEE; 2018. pp. 104–7.
URL: https://ieeexplore.ieee.org/document/8653728/. Accessed 4 March 2022.

6 Schneier B. Click Here to Kill Everybody: Security and Survival in a Hyper-connected World. First edition. New York ; London: W.W. Norton & Company; 2018.

7 Johari R, Kaur I, Tripathi R, Gupta K. Penetration Testing in IoT Network. In: 2020 5th International Conference on Computing, Communication and Security (ICCCS) [online]. Patna, India: IEEE; 2020. pp. 1–7.
URL: https://ieeexplore.ieee.org/document/9276853/. Accessed 6 March 2022.

8 Shahid A, Fontaine J, Camelo M, Haxhibeqiri J, Saelens M, Khan Z, et al. A Convolutional Neural Network Approach for Classification of LPWAN Technologies: Sigfox, LoRA and IEEE 802.15.4g. In: 2019 16th Annual IEEE International Conference on Sensing, Communication, and

Networking (SECON) [online]. Boston, MA, USA: IEEE; 2019. p. 1–8.
URL: https://ieeexplore.ieee.org/document/8824856/ . Accessed 30 March 2022

9 Sharma C, Gondhi NK. Communication Protocol Stack for Constrained IoT Systems. In: 2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU) [online]. Bhimtal, India: IEEE; 2018. pp. 1–6.
URL: https://ieeexplore.ieee.org/document/8519904/ . Accessed 20 March 2022

10 Barrett DJ, Silverman RE, Byrnes RG. SSH, The Secure Shell: The Definitive Guide. 2nd ed. Sebastopol, CA: O'Reilly; 2005.

11 Ylonen T. SSH — Secure Login Connections over the Internet. In: SSYM'96: Conference on USENIX Security Symposium: Focusing on Applications of Cryptography [online]. San Jose, California, USA: USENIX Association; 1996, Vol. 6. pp. 37–42.
URL:
https://www.usenix.org/legacy/publications/library/proceedings/sec96/full_papers/ylonen/index.html. Accessed 5 March 2022

12 Hoz JD de, Saldana J, Fernandez-Navajas J, Ruiz-Mas J, Rodriguez RG, Luna F de JM. SSH as an Alternative to TLS in IoT Environments using HTTP. In: 2018 Global Internet of Things Summit (GIoTS) [online]. Bilbao, Spain: IEEE; 2018. pp. 1–6.
URL: https://ieeexplore.ieee.org/document/8534545/. Accessed 6 March 2022.

13 Kerliu K, Ross A, Tao G, Yun Z, Shi Z, Han S, et al. Secure Over-The-Air Firmware Updates for Sensor Networks. In: 2019 IEEE 16th International Conference on Mobile Ad Hoc and Sensor Systems Workshops (MASSW) [online]. Monterey, CA, USA: IEEE; 2019. pp. 97–100.
URL: https://ieeexplore.ieee.org/document/9059463/ . Accessed 25 March 2022

14 Bauwens J, Ruckebusch P, Giannoulis S, Moerman I, Poorter ED. Over-the-Air Software Updates in the Internet of Things: An Overview of Key Principles [online]. IEEE Communications Magazine; Vol 58; 2020. pp.35–41.
URL: https://ieeexplore.ieee.org./document/8999425 . Accessed 25 March 2022

15 Frisch D, Reißmann S, Pape C. An Over the Air Update Mechanism for ESP8266 Microcontrollers. The Twelfth International Conference on Systems and Networks Communications [online]. Athens, Greece. IARIA XPS Press; 2017. pp. 12-17.
URL:
https://www.researchgate.net/publication/320335879_An_Over_the_Air_U

pdate_Mechanism_for_ESP8266_Microcontrollers . Accessed 16 March 2022.

16 Jaouhari SE, Bouvet E. Toward a generic and secure bootloader for IoT device firmware OTA update. In: 2022 International Conference on Information Networking (ICOIN) [online]. Jeju Island, South Korea: IEEE; 2022. pp. 90–5.
URL: https://ieeexplore.ieee.org/document/9687242/. Accessed 5 March 2022

17 Sharf SH, Elhamied RA, Habib MK, Madian AH. An Efficient OTA firmware updating Architecture based on LoRa suitable for agricultural IoT Applications. In: 2021 International Conference on Microelectronics (ICM) [online]. New Cairo City, Egypt: IEEE; 2021. pp. 262–5.
URL: https://ieeexplore.ieee.org/document/9664942/ . Accessed March 27.

18 UKEssays. Microcontrollers In Wireless Sensor Networks [online]. November 2018.
URL: https://www.ukessays.com/essays/computer-science/microcontrollers-used-in-wireless-sensor-networks-computer-science-essay.php?vref=1 . Accessed 27 March 2022

19 Espressif Systems, ESP32 Series Datasheet [online], Version 3.8; 2021.
URL:
https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf . Accessed March 28 2022

20 AZ-Delivery, ESP32 NodeMCU Module WLANWiFi Development Board with CP2102 Data Sheet [online].
URL: https://www.az-delivery.de/en/products/esp32-developmentboard . Accessed 15 March 2022

21 Merriam-Webster.com Dictionary, Merriam-Webster [online]. 2022.
URL: https://www.merriam-webster.com/dictionary/sensor. Accessed 28 March 2022.

22 Bosch, BME180 Digital Pressure Sensor, BST-BMP180-DS000-12 datasheet [online], May 2015.
URL:
https://components101.com/sites/default/files/component_datasheet/BMP180%20Sensor%20Datasheet.pdf . Accessed 29 March 2022

23 Elegoo Inc., 37 Sensor Kit Tutorial V1.0.18.10.25; 2022.
URL: http://69.195.111.207/tutorial-download/?t=37_in_1_Sensor_Modules_Kit . Accessed 28 March

Sensor Task Function

```
void Task_Sensor(void *pvParameters)
{
    (void)pvParameters;

    if (!bmp.begin()) {
       Serial.println("Could not find a valid BMP085/BMP180 sensor, check wir-
ing!");
       while (1) {}
    }

    while (1) // A Task shall never return or exit.
    {
        char tempall[60];

        DateTime.setServer("1.fi.pool.ntp.org");
        DateTime.setTimeZone("EET-2EEST,M3.5.0/3,M10.5.0/4");
        DateTime.begin();

        float temp = bmp.readTemperature();

        sprintf(tempall, "%s, %f \n", DateTime.format(DateFormatter::SIM-
PLE).c_str(), temp);

        File data_t = SPIFFS.open("/sensor_data.csv", FILE_WRITE);
        if (!data_t)
        {
          Serial.println("There was an error opening the file for writing");
          return;
        }
        if (data_t.print(tempall))
        {
          Serial.println("File was written");
        } else
        {
          Serial.println("File write failed");
        }
        data_t.close();

        vTaskDelay(600000);
    }
}
```