Evaluation of the ESP8266 as a Platform for Implementing Secured IoT Communications

Alex Hunter

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Bachelor of Science
of the
University of Aberdeen.

1 4 9 5

Department of Computing Science

2016

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: 2016

Abstract

The Internet of Things is now an ever-expanding collection of internet-connected smart devices that can transmit all kinds of information to their owners. As smart devices like these become more prevalent, sending out more and more information about their surroundings, it becomes increasingly important to ensure that this information is kept secure. When a house has been set up to send out temperature, electricity useage, or even CO2 concentration in each room, exposing this information to the wide world is practically like inviting them in (which isn't necessarily a good thing). If this issue is brought up to an industrial scale, things look even worse. Even so, we have the technology to prevent this from happening, and just about everyone uses it every day: TLS. We use it without knowing it when we click on certain web pages, so why can't we have such an easy-to-use solution for smart devices? A solution that doesn't require an in-depth knowledge of the protocol to get working?

This project aims to prove that this can be achieved, and assesses the suitability of the incredibly cheap and compact ESP8266 module as a platform for development and deployment of a simple TLS implementation. To do this, a simple TLS-PSK implementation is written, and its performance on the ESP8266 is evaluated.

Acknowledgements

I would like to express my thanks to everyone who helped me in bringing this project to fruition. In particular, thanks to Dr Martin Kollingbaum for his continued support, insight, and time spent figuring out the finer points of TLS with me. Additionally, thanks go to Dr Nir Oren for his very constructive comments, to Simon for listening to me babble about each week's new set of issues, and Nishad for helping to clear quite a few things up. Finally, I would like to extend my warmest thanks to Lili, Brendan, and Flora, for putting up with me for the past three months in such an understanding manner.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet of Things is a blanket term that refers to any physical object which has embedded electronics that allow it to connect to a network and exchange information over it. [1] The boards and WiFi modules that make up the Internet of Things can be found in a variety of environments: in homes, offices, and even industrial plants, and it appears that they are only going to become more prevalent as time passes [5]. As we pump more and more information about ourselves out onto the world wide web, it starts to become more and more important to secure this information so that prying eyes can't simply start reading it to gain an unwanted window into our homes and offices. This project aims to prove that our information can be secured, even on low-power WiFi chips. The ESP8266 is chosen as the platform, mainly because of its compact size and surprisingly low price point, which has been described as: "thousands of sensors-launched-out-of-a-cannon-cheap"[2]. More specifically, this project involves creating a proof-of-concept TLS-PSK implementation on the ESP8266, evaluating the performance of its parts, and then discussing the pros and cons of using the ESP8266 for such a project.

1.1 Motivations

The ESP8266 is looking to be a very promising chip for embedded systems. Its low cost paired with its compact size make it ideal for implementing both large-scale sensor networks, and small-scale home projects. This project aims to explore the most important advantages and shortfalls that arise from using this chip in place of other, more expensive hardware, such as the Raspberry Pi. Additionally, the lack of built-in features for secure communications means that users either need to send unsecured messages, or get to know more complicated libraries such as mbedTLS. At present, existing TLS libraries for the ESP8266 aren't nearly as simple as browser-based TLS (which is so simple from a user perspective that one could be forgiven for not even noticing it). To address this, this project aims to produce an implementation that allows users to secure their communications with as few modifications to their original code as possible.

1.2 Goals

The main goals of the project are the following:

[1] `www.https://en.wikipedia.org/wiki/Internet_of_Things`

[2] `https://learn.adafruit.com/esp8266-temperature-slash-humidity-webserver/overview`

Implement Secure Communications on the ESP8266

There is currently no built-in TLS support, likely due to the hardware limitations of the chip. A TLS-PSK implementation will be made in order to demonstrate the capabilities of the ESP8266 in terms of securing communications.

Simplicity

This implementation should be as easy-to-use as possible: the amount of interaction the user must have with the implementation is to be comparable to that of TLS in a web-browser.

Evaluate Implementation and Platform Effectiveness

Evaluate the quality of the implementation, as well as the effectiveness of the ESP8266 as a platform for said implementation.

Comparison of ESP8266 with other Hardware

The ESP8266 is not the only device used for IoT systems. This section provides a comparison between it and some other devices used for similar purpose in a cost-benefit fashion. In particular, performance, development process, and costs will be evaluated, relative to the alternative devices.

Chapter 2

Background

This chapter provides background information about the topic of this project.

2.1 IoT WiFi Sensors

As a concept, the Internet of Things can more accurately be described as ordinary objects being fitted with sensors, actuators, or both, which are connected to the internet so that they can interact and be interacted with remotely. We can install sensors to measure temperature, sound, light, among other things, as well as actuators like motors to bring about physical movement like the opening of doors. The one requirement that turns these devices into something that can be considered part of the Internet of Things is a means of communicating via the internet. This is where things start to get complicated: we need chips that are small, energy-efficient, and which have enough computational power to interpret and send data in the right fashion at the speed required. There are already some areas in industry that could benefit from the addition of cheap IoT sensors like these. For example, vaccines are required to be stored at a particular temperature range and, if they are not, they can become compromised. This presents a danger, as people receiving these ineffective vaccinations would be unaware that they are still at risk of catching some dangerous diseases, and may cause this to happen. [6] With this in mind, it becomes important to ensure that vaccines and medicines with similarly temperature-sensitive properties are constantly kept at the right temperature. While this can be observed manually, this might require round-the-clock supervision, in order to address issues such as power cuts. However, if battery-powered temperature sensors were to be deployed, they could be set to monitor the temperature in the room, and even send alerts out to the appropriate people if the temperature were to drop. Even better, they could be linked to actuators that control the temperature, allowing a cheaper self-regulating system that allows monitoring and stabilization of the temperature. Another, more simple example of such things would be Amazon's Dash button. This is effectively a WiFi enabled button that sends a predefined message to Amazon's servers via WiFi upon the button being pressed, with an aim to allow users to quickly "restock" certain products. [7] These are just 2 examples of potential and existing markets for these devices. The scope of where they can be implemented is only really constrained by cost and availability of the components.

2.1.1 Boards for IoT

The key link that turns a standard sensor or actuator from just that into an IoT-connected device is the ability to communicate via the internet. This is can be achieved with a WiFi chip of some description. In the past, boards such as the Roving RN-131 could be used, as well as the Raspberry

Pi 3 and Arduino Uno. However, the Raspberry Pi and Roving boards generally cost in excess of £20 per unit, with the Arduino boards often sitting around the £10 mark. While these prices are probably affordable for most small-scale home projects, an industrial setting that could require large numbers of sensors would soon start to feel the costs of these. It is worth noting that there are many factors that influence how useful a board is. Some significant factors to consider would include the maturity of the environment, which would influence available libraries and amount of support available from the community, and ease of development, which is influenced by the number of options available for developing for a given platform, as well as the nature of these options. For example: the Raspberry Pi supports most languages that work on Linux machines, including Python, C/C++, Java, and Ruby [8] whereas ESP8266 boards mainly support C++, or can be flashed with NodeMCU firmware to support LUA scripting [9].

2.1.2 The ESP8266

The ESP8266 is a very cheap alternative to these boards. Its main selling point is that its price is a fraction of what any of the previously mentioned boards is, while still offering enough computational power to perform most tasks expected of embedded devices. It first appears to have surfaced in August 2014, when it was considered to be a component that could be paired with other micro controllers to enable them to access the internet. [10] In the months that followed, it was found that the on-board microcontroller of the ESP8266 could itself be programmed, removing the need to pair this device with another, and thus drastically reducing costs and power requirements. The situation was further improved by the manufacturer, Espressif, releasing its SDK, after which, modules for the Arduino IDE were created that implemented ESP8266 support. [25] This has led to the development of communities such as the Everything ESP8266 message board, which allows users to share experiences and questions with one another. At present, many Arduino libraries have been ported to the ESP8266, including most of the WiFi related code. However, there is a clear lack of support for secured communications (more on this later).

2.2 Internet Security

Information security is vital in allowing users to safely connect to the internet with minimal risk of unauthorized access to their data. Securing communications helps to prevent third parties from accessing sensitive information, such as bank details and personal information, such as name and address.

2.2.1 Transport Layer Security

One of the most widely used protocols that provides assurance that data is kept secure is the Transport Layer Security protocol - TLS. Previously known as SSL (Secure Sockets Layer), TLS allows servers and clients to confirm each others' identity, before transmitting encrypted data which can only be decrypted through the use of some shared secret key. [22] The main functions of TLS are to prevent unauthorized access to or modification of data sent and to authenticate the server to the client. TLS may also provide data integrity through means of a Message Authentication Code (not available for all ciphers) and authenticate the client to the server, although this is not usually done, as the client can usually authenticate by some other means once a TLS session has been established.

Sessions are established through a protocol called the TLS Handshake. This handshake involves both parties exchanging random key material in the clear, before verifying each others' identity by sending and verifying certificates. Both parties then use information which has been acquired from previous messages and certificates to create asymmetric keys, before using these keys to encrypt some information that will allow the calculation of some symmetric keys. Asymmetric keys refer to the keys used for asymmetric encryption algorithms. These algorithms are used so that data encrypted with one key (typically the public key) can only be decrypted by applying the decryption algorithm to the cipher text with the private key. The symmetric keys are used by symmetric algorithms for both encryption and decryption. These algorithms use the same key for encryption and decryption. [11] Under TLS, symmetric algorithms are generally used for bulk encryption of data and asymmetric algorithms are used to allow both parties to secretly agree on a symmetric key to use. Much can be said about these algorithms, and the reasons for each being used; however a variant of TLS called TLS-PSK is used in this project in order to avoid the computational overheads of asymmetric cryptography. This version of the protocol simply assumes that the client and server already know the symmetric encryption key, and thus, do not need to do much in the way of key agreement. Following the handshake, all communications between the two parties are encrypted and decrypted under the symmetric encryption algorithm.

2.2.2 TLS-PSK

This project uses a type of TLS called TLS-PSK, short for TLS- Pre-Shared Key, which allows the server and client to communicate without the need for expensive key establishment algorithms. The server has a list of pre-computed keys, each of which is attached to a unique ID which would usually correspond to a particular client (although this is not a necessity). The TLS handshake for this protocol is markedly different from the standard handshake protocol. The protocol is outlined in Figure 2.1. This figure both shows the TLS-PSK handshake and displays the differences between the TLS-PSK handshake and a regular TLS handshake (see caption for details).

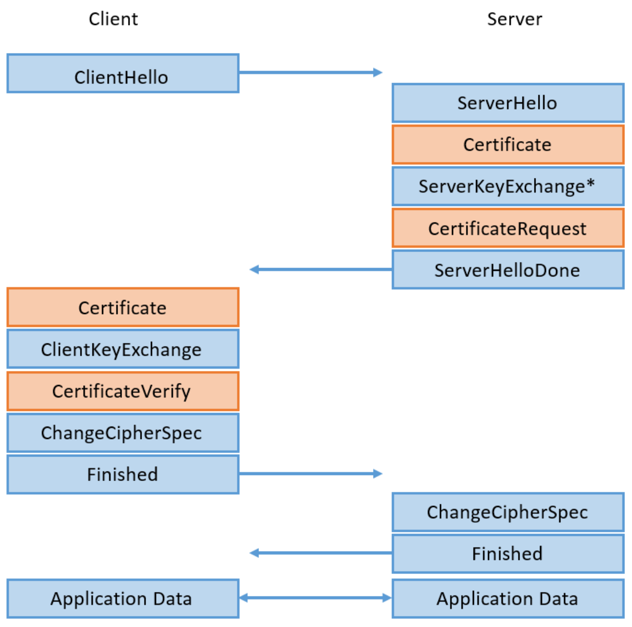


Figure 2.1: The TLS-PSK handshake, as described in RFC4279[4]. Messages which would occur in a normal TLS handshake, but not in a TLS-PSK handshake are highlighted in orange

2.2.3 The TLS-PSK Handshake

The TLS record protocol dictates that all TLS messages contain information which denotes the TLS version (1.0, 1.1, or 1.2 are the possibilities at the time of writing), as well as the content type, which can be one of: Handshake, Alert, ChangeCipherSpec, Application Data or Heartbeat. Handshake protocol messages also contain a field which denotes which represents the handshake message being sent.

The ClientHello message contains the appropriate information as previously stated, alongside client random, which is the 4-byte representation of the current Unix time concatenated with 28 random bytes. This information is used later in determining secret shared data between the client and the server. Following client random, this message also contains a session ID (if resuming a session), a list of supported cipher suites, a list of compression methods, and any optional extensions.

The next message to be sent is the ServerHello message, which contains all the fields that ClientHello contains, with a few differences. The main differences are that the cipher suite and compression method fields are now comprised of one cipher suite and one compression method respectively; this is because the server will choose the best option for each and transmit its choice to the client.

This message is (optionally) followed by a ServerKeyExchange message, which simply provides a PSK identity hint. This field is quite self-descriptive: it provides the client with a hint from which it can either calculate or choose a pre-shared key identifier to use. Each symmetric key that the server holds has one PSK ID associated with it, and the client must select the PSK ID that corresponds to a key that both itself and the server have in common. This message is not mandatory; it could be assumed that the client requires no hint.

Again, this message is immediately followed by a ServerHelloDone message, which indicates that the server has finished its “hello” phase of the handshake.

The client must then respond with a ClientKeyExchange message, which generally contains the PSK ID of the PSK that it intends to use with the server.

This message is then followed by a ChangeCipherSpec message, which signals that messages from then on will be encrypted.

The next message is the Finished message, which contains a hash of all messages sent up to this point. This hash is created using a Pseudo-Random Function, which uses information derived from the client and server’s random data, as well as the PSK being used, as key material (this process is detailed in the “Implementation” chapter). This effectively serves as a Message Authentication Code (MAC) for the entire handshake exchange.

Following the handshake, the PSK is used by both parties to encrypt application data in the same way as regular TLS.[4]

2.2.4 Local Security

An even more common security measure that can be seen in the vast majority of households is that found on most routers. These will generally use one of a number of Pre-Shared Key protocols (such as WPA-PSK). The function of this is to encrypt communications between devices connected to the router (or other access point) and the router/access point itself. Unfortunately, there are still vulnerabilities in this system. Chief among them is a vulnerability similar to “the Chromecast

Vulnerability". This vulnerability results from an issue with the WiFi standard which involves the "deauth" command, which tells the device that it needs to disconnect from the network and attempt a reconnect. However, this command is sent unencrypted, so any device could send this message, assume the same name as the original access point, and have the target device connect to it instead. This could lead to all unsecured communications between the client and server being readable by some third party.[12] This vulnerability also applies to the ESP8266. In fact I, personally, was able to confirm this using a modern laptop and the access point feature of a mobile phone. However, this vulnerability does not completely compromise the security of a device: some cipher suites in TLS protect against man-in-the-middle attacks that would be made feasible in this way, so even if a third party was sniffing packets, they would still have no way to decrypt the communications. The only real effect of this would be that the third party could determine when the device and server are communicating, or it could simply not send any communications at all, effectively blocking the device from communicating with the server.

Chapter 3

Related Work

Work relevant to the topics of this report is discussed in this chapter. Work done to evaluate different approaches for implementing WiFi sensors is examined, as well as research which addresses the feasibility of implementing security protocols in constrained environments. Additionally, similar implementations of security protocols for embedded devices are mentioned, as well as their differences from the implementation described by this report.

3.1 Internet of Things WiFi Sensors

It appears to be generally accepted that the Internet of Things is becoming more and more of a reality. [13] Given this, it is important to examine the performance of the WiFi module to be used. This is because these modules vary in performance, energy efficiency, and price, and which combination of these 3 is best will likely be largely dependent on the situation. It has been stated that security architecture design choices are dependent on the intended lifecycle of the device [14], and this is no less true for other factors influencing choices relating to the system as a whole. In particular, the Roving RN-131 has been evaluated as being sufficiently energy-efficient to run for a reasonably long time, while connecting to WiFi periodically to send data. [13] However, other research also takes into account the overheads that different types of security implementation can have on this. In particular, the choice of authentication for communicating with an access point (such as WPA, WPA2, and WEP) is said to have some effect on this, particularly for devices that function in a "sleep and wake" fashion; entering a low-power state, before powering up to retrieve data, connect to the access point, and send said data [15]. IP-layer security is also a major concern. This is discussed further in a later section.

3.2 The ESP8266

The ESP8266 is a cheap WiFi module which ships with the complete AT command set. The module is perfectly capable of simultaneously hosting and receiving data over wireless connection, and can be very easy to use with other hardware.[16] However, the device really becomes interesting when the original firmware is overwritten, in favor of code which is custom-made by the user. This turns the ESP8266 from an accessory to be used to add wireless capabilities to other micro-controllers, into WiFi-enabled micro-controller able to rival existing micro-controller unit (MCU) boards (such as certain Arduino micro-controllers) in terms of computational power. The ESP8266 can be flashed using the Arduino IDE, which uses a cut-down version of the C++ compiler (avr-gcc) to create "sketches", (basically small C++ programs) that can be flashed onto

Table 3.1: Comparison of ESP8266 and Similar Boards [1] [2] [3]

	Arduino Uno	ESP8266	Arduino Due
CPU Clock Speed	16 MHz	80 MHz	84 MHz
Flash Memory	32 KB	512 KB	512 KB
SRAM	2 KB	36 KB	96 KB
Price	$6.95	$24.95	$49.95

the module in place of other firmware. The ESP8266 is comparable to certain Arduino boards, and appears to sit in between the Arduino Uno and Arduino Due boards in terms of its computing resources. Table 3.1 shows a comparison of some basic specs, alongside the prices for each board.

3.3 The IoT Security Lanscape and Requirements

At present, Open Web Application Security Project (OWASP) considers the lack of transport encryption to be the 4th most significant vulnerability in the Internet of Things[17]. The general consensus has been, over the past few years, that there is no commonly accepted standard for Internet of Things security.[18] This is likely due to the broad range of hardware that is encompassed by the IoT. This can include anything from the 16Mhz Arduino Uno to the 1.2GHz quad core Raspberry Pi 3. When designing a security architecture, a number of factors need to be considered, including how scalable the system must be, the lifecycle of the system, and any constraints imposed by the hardware. [14] The main limiting factors of hardware, in this context, tend to be processing power and available RAM, with RSA decryption taking around 88 seconds for 1024-bit keys, and 701 seconds for 2048-bit keys on an Arduino Due. [19]

Another caveat of introducing expensive asymmetric encryption operations is the amount of battery power being drained over the great length of time it takes to do the calculation. These devices tend to use more power when in active mode relative to when they are in sleep mode. Sethi et al. [18] used an Arduino ATmega2560, which uses 20mA in active mode and 5.4mA in sleep mode. Their results showed that the time to decrypt a message under RSA with 2048-bit keys was at 1,587,567 ms if keys were stored in SRAM, and 1,740,258 ms if keys were stored in RAM. This is roughly equal to 26 or 29 minutes of time solidly spend doing the decryption. Results at 1024 bit key size were more encouraging; both were around 200,000 ms, which translates to between 3 and 4 minutes. Whether or not this is acceptable depends entirely on the context: if one TLS handshake involving RSA is required before a constant stream of data is sent, and high security is required, this would be an acceptable time. However, sensors that sleep and reconnect would likely need to re-do the handshake each time. In this scenario, the RSA decryption time would become a severe bottleneck. The paper does later state that alterations to the algorithms could be made in order to better suit embedded devices. A thesis published in 2006 partly addresses this by including some useful recommendations when designing implementations of cryptographic algorithms [20], but that is beyond the scope of this report.

3.4 Similar Work

There are some existing implementations of TLS which can, in theory, run on any embedded device. A good example is MbedTLS; an implementation of TLS originally called PolarSSL, this library is maintained by ARM mbed and is designed to be used on embedded MCUs, such

as Arduinos. [21] mbed TLS is a fairly large implementation, and using it does require some knowledge of how TLS works. One of the aims of this report (elaborated upon later) was to provide an implementation of TLS that is as simple to use as possible, from the developer's standpoint.
Another implementation is the much more well-known OpenSSL. While this could be considered a good candidate, it is not optimized for embedded devices, as it proved very difficult to get it working on the ESP8266.

Chapter 4

Requirements

4.1 Functional Requirements

4.1.1 Investigate the Performance of the ESP8266 when Sending Data Over an Unsecured Connection

Create implementation with no real IP security features so that we have benchmark performances against which to compare implementations with IP security features included. This is important in informing users when deciding which security implementation works best for them. Ideally most users will have some form of security since, even though the information itself might not be sensitive, it could still be modified by a well-placed man-in-the-middle attack.

4.1.2 Build a Basic TLS-PSK Library

A TLS-PSK library should be created so that it can be used, both to secure communications, and as a means of testing the ESP8266' ability to do so. Ideally, the implementation should be usable afterwards, but its purpose, in this report, is as proof-of-concept software for evaluating the ESP8266.

4.1.3 Evaluate the Performance of this Library

Data should be sent securely, using this library, and the time taken to do so should be compared to the time take to do so when the information is not secured. Individual modules of the implementation will also be tested in order to identify any areas that cause particular bottlenecks, or which cause stability issues.

4.1.4 Capture and Analyze Packets to inform the Design

Packets containing messages sent to and from the server should be collected in order to inform development decisions; errors and alerts being received from the server would indicate that something is wrong either with the implementation, or the server.

4.2 Non-Functional Requirements

4.2.1 The TLS-PSK Library must be Simple and Compact

The TLS-PSK implementation should be as simple to use as possible. This requirement arose from the observation that existing TLS implementations, such as mbed TLS and OpenSSL still require a reasonably good knowledge of the protocol. This implementation is to be designed in a “plug and play” manner, allowing the user to achieve some degree of security easily.

4.2.2 Adhere to the Relevant Standards for TLS

It can be assumed that most servers that claim to have TLS will be adhering to these standards where possible. This implementation should also adhere to them as closely as possible, in order to be compatible with other implementations. The standards being adhered to will primarily be those specified in the relevant RFC documents, particularly RFC 4279[4] and RFC 5246 [22].

4.2.3 Only use External Software when it is Free and Modifiable

Many existing resources will not compile on the ESP8266, since they may not be designed with the AVR compiler in mind. This means that software that is to be used must be modifiable in order to function on the module. An example of this is the AES module which was written in C, but which needed to be altered in order to work as part of a C++ program. Additionally, only using free and open source material means that this implementation can be freely distributed without the need to pay any fees.

Chapter 5

System Design

This section explains and justifies the more general design choices made over the course of the project. This includes the test bed, the choice to use TLS-PSK, and software required to run tests.

5.1 Test Bed

The same test bed was used for the development of most aspects of the implementation. It involved a PC being connected to an Olimex ESP8266-EVB via a USB to serial converter. The ESP8266 board was powered by a 5v DC power supply. The server and access point for the ESP8266 was a Raspberry Pi 2 model B connected to a wireless router via Ethernet. The Raspberry Pi also had a USB WiFi module connected. This allowed it to serve as an access point, to which the ESP8266 would connect and send data. The Raspberry Pi also functioned as a server, which hosted a simple web page that displayed a table of the last 10 records of information stored in a database. This table was updated continuously using javascript, so that additions could be observed in real-time. Additionally, the Raspberry Pi served as a means of packet capture: tshark, a command line utility for Wireshark, was used to capture any packets being sent or received from the wireless interface. This meant that it was able to capture packets sent by the ESP8266 during connection to the access point, as well as packets containing either data, or TLS messages. Finally, a piece of software called Stunnel was installed on the Raspberry Pi. This is a tunneling service that can be used to provide TLS functionality to non-TLS enabled servers. The reason for its use was that Apache does not seem to support TLS-PSK natively, whereas Stunnel appears to. Figure 5.1 illustrates this architecture.

An Olimex ESP8266-EVB board was used for testing purposes. This board is a somewhat large breakout board that allows the board to be reset and reflashed more easily than some other models, such as the ESP8266-01. For implementation, ideally a smaller board would be used, as there would ideally not be a need to frequently reprogram it.

5.2 Software Components

A number of programs needed to be installed on the Raspberry Pi so that it could carry out its functions as a server, access point, and packet sniffer.

5.2.1 Apache 2

Apache 2 was used to host the server. Apache is the most frequently used piece of software (on Linux) that allows the hosting of web servers. [23] Initially, the SSL functionality of Apache was enabled, but it was later discovered that it does not support TLS-PSK.

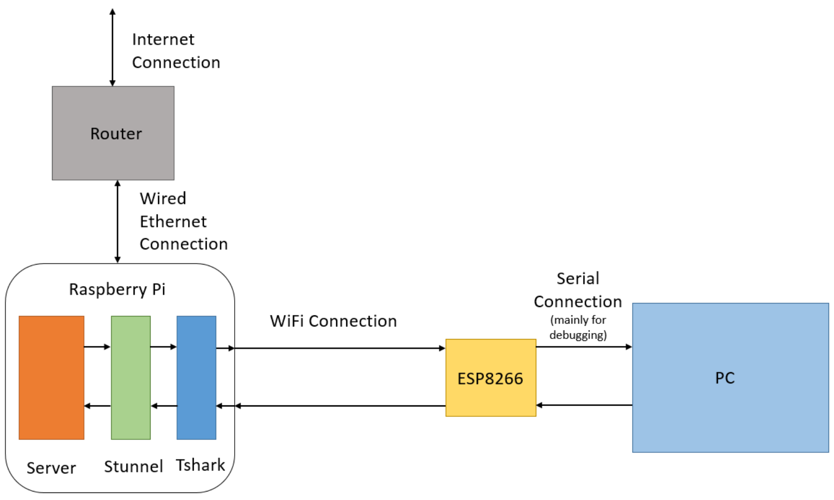


Figure 5.1: The architecture of the test bed. The Raspberry Pi has an Apache 2/PHP 5 server hosting on port 80, Stunnel listens on the HTTPS port 443 and forwards to the HTTP port 80, and Tshark records all packet data.

5.2.2 Stunnel

As a result of this, stunnel was used, since it claims to support TLS-PSK. Stunnel is a tunneling engine that allows servers that do not have TLS enabled to be secured with TLS. It works by taking traffic from the TLS/SSL port, performing all TLS-related operations (handshake, decryption/encryption, etc), and passing on any relevant decrypted messages as regular, non-TLS packets to the server which is listening on some predefined port (80 by default, but maybe some other port if one wants the site to be more difficult to access unsecurely).

5.2.3 Hostapd

Hostapd (which stands for Host Acces Point Daemon) is sofware that allows a linux machine to function as a network bridge. A network bridge takes network traffic from one source (an interface, such as an ethernet port or USB WiFi module) and forwards it to another. In this scenario, the Raspberry Pi had a USB WiFi module connected to it which allowed wireless communication with the ESP8266, and an ethernet cable connecting it to an internet-connected router. Traffic would be sent from the ESP8266 to the wireless module, and it would be forwarded by Hostapd to the Ethernet interface. The purpose of doing this, rather than simply connecting the ESP8266 directly to the router was so that the packets being sent to and from the ESP8266 could all be monitored using Tshark (described in section 7.2.3). Even if capture software could be run easily on the router, there was consistently a lot of background noise: thousands of packets would be captured in the space of a minute, due to the number of other users, so sifting through those would create unnecessary extra work, whereas the only device connected to the Raspberry Pi access point was the ESP8266.

5.2.4 PHP/JavaScript/MySQL

The web site on the server consisted of a MySQL database, PHP scripts, and an HTML file containing javascript. The PHP script was used to establish a connection to the database and to populate it when receiving temperature reading data from the ESP8266. The database was used to store temperature data readings. The javascript was not completely necessary, but allowed the data readings being added to the database to be observed approximately in real-time.

5.3 TLS-PSK

This section describes design choices related to the TLS protocol used. In particular, the choices to use TLS-PSK, and a particular cipher suite within this protocol, are explained.

5.3.1 Why TLS-PSK?

TLS-PSK was used as a starting point because it was the simplest way of proving that some TLS implementation could work. Additionally, it appeared to be the best way of avoiding expensive asymmetric cryptography calculations, as these can take some time for a small device such as this [18]. TLS-PSK works in a similar manner to regular TLS, only no certificates need to be exchanged. This is useful for memory-constrained environments such as IoT devices because certificate chains and stored certificates can be multiple megabytes in size, which is significantly more than even the total amount of RAM and flash memory on the ESP8266 (as seen in chapter 2). As mentioned, Asymmetric cryptography can take a long time on boards like the ESP8266 (in the order of minutes in some cases), so removing this overhead from both the development loop, described in chapter 7, and from the program itself at runtime, is advantageous.

5.3.2 The Cipher Suite

The cipher suite used for this implementation is TLS_PSK_WITH_AES_128_CBC_SHA256. To break it down, this cipher suite specifies that TLS-PSK will be used, that the bulk encryption cipher is AES 128 in Cipher Block Chaining (CBC) mode, and that the hash algorithm will be SHA 256. AES-128 was selected for a number of reasons. The main reason was, frankly, that AES appears to be the most popular encryption algorithm, in terms of the number of implementations compatible with the ESP8266. The only reason that 128-bit encryption was used instead of a higher number was that, of all the implementations tested, a 128-bit implementation was the only one that could be made to work on the ESP8266 (although only with heavy modification. See chapter 7 for details). Additionally, AES-128 performed very well in terms of time taken to encrypt and decrypt text, so there was no need to look for anything more efficient. SHA-256 was chosen, again, because it was part of a readily available cryptography library. It appeared to be the strongest library available, and its performance was good enough that it didn't become a bottleneck to the speed of the handshake. SHA-1 could have been used, as it was available in the same library, but since the SHA-256 implementation performed adequately, it was decided that the improved security was more beneficial than the minute amount of time saved by using the weaker algorithm.

Chapter 6

The TLS-PSK Protocol

This section describes the TLS-PSK protocol in detail, so that the description in the "Secured Implementation" section of chapter 7 can be more brief. It describes what information is carried by each packet, although the exact positioning and size of each field of the packets is omitted, as this information can be found in the relevant RFCs and Wikipedia entries. CITATION

6.1 The Record Protocol

The TLS record protocol is the same across all cipher suites. It specifies how to send any and all TLS packets so that the content type, TLS version, and length of the packet can be determined. The first field of all TLS packets represents the content type of the packet, which can be one of: Change Cipher Spec, Alert, Handshake, Application, or Heartbeat. This is followed by the version field, which specifies the TLS version to be used. Currently, the only version considered secure are SSL 3.0, TLS 1.0, 1.1, and 1.2. After this comes the length field which represents the length of the entire TLS packet, which may include multiple TLS messages. These can be of different content types, but multiple messages of the same type can also be sent. The purpose of this field is to allow any devices parsing the packet to know when this packet ends and, potentially, where another begins (in case another is sent immediately after). Finally, the protocol messages appear, one after the other. All message types will first specify the content type, TLS protocol version, and length of the message. Following this, the next field(s) are dependent on the message type and content type.

6.2 The TLS-PSK Handshake

TLS handshakes that use different cipher suites can have significant differences in terms of the messages transmitted. This is because the underlying algorithms require different information. All TLS handshake messages will be contained in a record with the previously stated structure. They will then display the content type (handshake message), TLS protocol version, and the length of the message. All fields following these are, from this point in the document, referred to as "inner message fields" for brevity.

Client Hello

The client hello message is sent from the client to the server in order to initiate the handshake. It is largely the same for all cipher suites. The first inner message field is client random, which is 4 bytes representing a Unix epoch time (the number of seconds since midnight, January first, 1970), concatenated with 28 bytes of random data. The next field consists of pairs of bytes representing the supported cipher suites concatenated into a long list. The length of this list is specified. In a similar fashion, there is a list of supported compression methods (note: this can be, and often is, null).

Server Hello

After receiving a Client Hello packet, the server can either respond with a Server Hello message, or an Alert message. An alert message will be sent if, upon parsing the cipher suites field of the client hello packet, it finds that it does not support any of the listed suites. If it finds suites it supports, and the packet is correctly composed, the Server Hello message is sent. The purpose of this is to tell the client which cipher suite and compression method are to be used, The first inner message field is server random. This follows the same format as client hello: 4 bytes for the Unix time followed by 28 securely generated random bytes. Following this is the session ID. This field is only non-empty if a session is being resumed, in which case the ID of a previous session will be in this field. Finally, two fields, each representing the chosen cipher suite and then compression method are found. In both cases, the server will select a cipher suite from the list given to it by the client, provided that at least one of them is supported.

Server Key Exchange

This message is optional, and is only sent if the server is configured to send a PSK identity hint. When a server is configured for TLS-PSK, it must have a list of PSKs that can be used. Each PSK must have an associated identity, which is a string that effectively functions as the PSK's index. The identity hint's purpose is to indicate which PSK the client should request. How this hint is used is completely application dependent. This message is, in the case of TLS-PSK, exclusively used for sending the PSK identity hint, and thus, the only inner message fields are the length of the hint, followed by the hint.

Server Hello Done

At the end of the Server Hello phase, this message is sent. Its purpose is simply to indicate this fact. It contains no inner message fields (i.e. the outer length field is 0 and the message ends).

Client Key Exchange

In a similar manner to the Server Key Exchange message, this message simply send the PSK identity that corresponds to a PSK that should be used for communication. The inner message fields are the PSK identity length and the PSK identity.

Change Cipher Spec

This message has a different content type field to other handshake messages, however it is still part of the TLS handshake. The purpose of this message is to state that all communications following it will be encrypted in some manner. It contains no inner message fields, much like the Server Hello Done message. It is sent by both parties immediately before sending their respective Finished messages, as Finished messages do not contain plain text data.

Finished

The client version of the Finished message is the same, on the surface, as in the regular TLS handshake. The only field it contains is a hash of all the previous messages sent in the handshake. This hash is created using a function called the Pseudo Random Function (PRF). This is further discussed in the Implementation chapter. Upon receiving this message, the server will apply the same PRF to a hash of all messages before this one, and check it against what was received from the client. This is effectively a checksum operation, and guarantees the integrity of the handshake. Integrity is guaranteed because, if a message from the client had been modified after it had been sent, the client's resulting hash would be different to that of the server. Once this is done, the server sends a Finished message, containing a hash of all previous messages, including the client's Finished message. The reason for including this last message is so that the server cannot simply send back the same hash that the client sent. This verifies the integrity of the handshake for both parties. As discussed in a later chapter, the reason that the hash is secure is that the PSK is needed to generate a parameter for this run of the PRF, so a third party would need to know the PSK in order to be able to recreate it.

6.3 TLS-PSK Application Data

TLS-PSK application data is exactly the same as that sent by other protocols. Its only inner message field is encrypted application data. In this case, it is encrypted using the PSK.

Chapter 7

Implementation

The final product consisted of an example sketch alongside an external library that contained 2 class files and 2 header files. One class/header file pair was for the TLS implementation, and contained a class with methods relevant to performing the TLS handshake, and sending application data. The other was an AES128 CBC implementation that had been converted from C to C++. Before being added to the TLS class file, methods were gradually built within the sketch, for the sake of simplicity. Once the methods were complete and working, they were moved to the library and tested again.

7.1 Development

7.1.1 Process

The development of this project was was primarily an incremental development process. The technical requirements were mostly defined by the relevant RFCs, although other requirements such as fault tolerance and readability of code were planned ahead of time. Each step of the development process consisted of a loop of actions. Figure 7.1 illustrates this loop.

Each method was relevant to a new phase of the TLS protocol. For example, composing the ClientHello message and sending it was the first step. This was then followed by receiving and parsing the ServerHello message, and so on. Utility methods, (i.e. methods repeatedly used by others, such as the sendPacket() and receivePacket() methods) were developed as and when they were needed.

7.1.2 Challenges

There were a number of obstacles to development that are inherent to the platform. One issue that became apparent very quickly was the amount of time it takes to complete one development loop (shown in Figure 7.1). In particular, the compilation and upload process on Arduino IDE could be fairly time consuming. The time taken to compile the project would take longer and longer, sometimes taking up to 10 minutes. This time would increase with each compilation (although restarting Arduino IDE did not solve the problem - a full system restart of the PC was required to reset the compilation time). Unfortunately, the solution of restarting was only discovered approximately midway in the development process.

Another obstacle was the slight immaturity of the environment, relative to that of the Raspberry Pi or other devices. While there are very helpful and useful communities to refer to, there were not many relevant libraries developed specifically for the ESP8266. While there were some developed for embedded devices, it turned out that a surprising number of them would either not work on

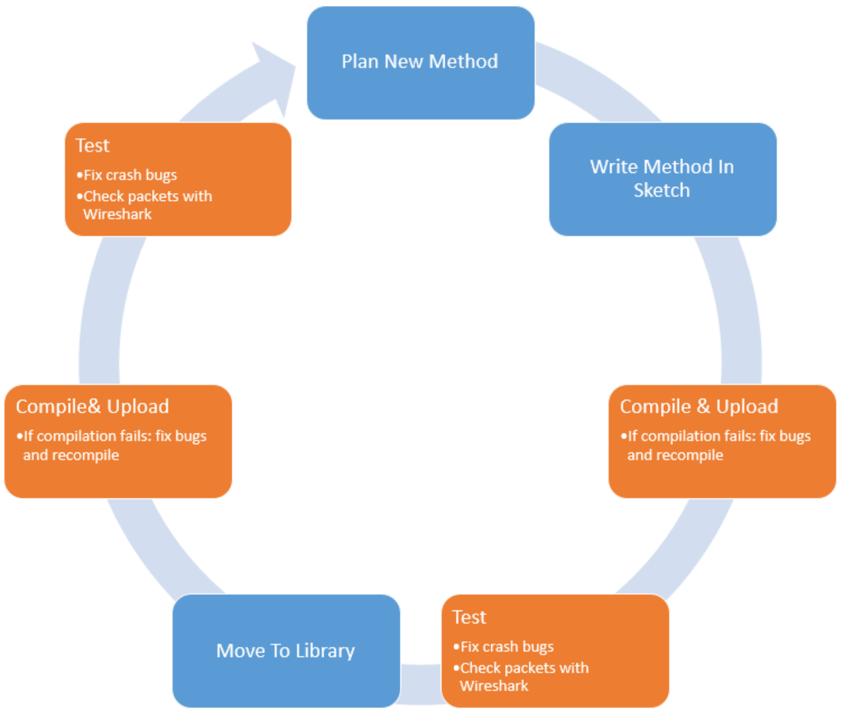

Plan New Method
Write Method In Sketch
Compile & Upload
•If compilation fails: fix bugs and recompile
Test
•Fix crash bugs
•Check packets with Wireshark
Move To Library
Compile& Upload
•If compilation fails: fix bugs and recompile
Test
•Fix crash bugs
•Check packets with Wireshark

Figure 7.1: The main development cycle.

the device at all, or would require some significant modification to do so. An exception is the SHA/SHA-256 library, however an AES library from the same developer would not run on the board.
The reason that some C++ libraries cause issues with the chip is that the Arduino IDE does not support some standard C++ libraries (for example, the new and delete keywords do not work, so these need to be replaced with similar calls to malloc() and dealloc() respectively). This is because it uses the avr-gcc compiler. Additionally, this compiler does not, by default, support try/catch blocks, or the STL library (although a port of this was found and used in order to be able to use vectors)[24].

7.2 Tools

A number of tools were used to enable and support the development process. This section discusses these tools and their uses.

7.2.1 Arduino IDE

The Arduino Integrated Development Environment was used to write and upload code to the ESP8266. It was initially intended to be used with Arduino boards, but has since had support for the ESP8266 added [25].

7.2.2 Wireshark

Wireshark is a tool that allows the capture and analysis of network packets from a wide range of network protocols. In this project, its purposes were twofold: first, it was used to capture TLS packets from regular network traffic, so that they could serve as examples for the composition of similar TLS-PSK packets in the project. Second, it was used to analyze .pcap files created by capturing and saving the network traffic going to and from the ESP8266. This was done to verify that the packets were composed correctly (e.g. to check that the length fields accurately reflected the amount of data following them).

7.2.3 Tshark

Tshark is effectively a command-line based version of Wireshark. It allows the capture and analysis of packets in a similar, albeit slightly less user friendly, manner to Wireshark's GUI. It was used primarily for filtering and capturing packets on the Raspberry Pi. Since the Raspberry Pi functioned as both the access point and server, it was assumed that all packets relevant to TLS-PSK could be captured.

7.2.4 GitHub Desktop

GitHub desktop was used for version control. GitHub desktop is a Windows application that provides a user friendly interface that allows the creation and management of Git repositories. It appears to offer more usability, but less fine-grained control over the repositories. It was decided that this was a worthy compromise, as I was the only contributor to the repository, so few functions beyond updating and rolling back were requred. A private repository was created to house relevant sketches, and the TLS library file was periodically backed up to this repository.

7.3 The Unsecured Implementation

Before an attempt to secure the communications between the temperature sensor and server could be made, an unsecured version of the program needed to be built. This was so that there was a benchmark for the speed at which data can be sent to the server. This sketch was effectively a modified version of an example sketch for sending data to sparkfun.com. The main modification to the sketch was the addition of the "connectWiFi'()" method, which allowed the device to reconnect to the access point if it disconnected. The other two methods are methods required by Arduino sketches, namely setup() and loop(). Elements from this implementation were later taken and put into a sketch that would send packets in a secure and insecure fashion, timing the time taken for each and comparing the average of 100 runs (this is discussed further in Chapter 8

7.3.1 Setup Function

This serves as an initialization function. It is called once automatically, and is used to initialize variables and pin modes, as well as initializing the serial output at a particular baud rate. In this sketch, it was used to connect to the access point for the first time, and initialize the serial output[26].

7.3.2 Loop Function

The loop() function, unlike the setup() function, is run continuously. Its main function is to control the board[27]. In the context of this sketch, this function contains the code that checks the status of the WiFi connection, reconnects if necessary, and then takes a temperature reading. It then puts the temperature reading into a string that is sent as a GET request to the server. This was sent by creating a WiFiClient object and calling the "connect()" method with the parameters being the address of the server and the HTTP port (80). Following this, the "print" method was called. This method sends a string to the server in plain text. Following this, the "readStringUntil()" method was called to retrieve any responses from the server.

7.4 Board Setup

The ESP8266 was connected to a DS1820B temperature sensor as shown in Figure 7.2.

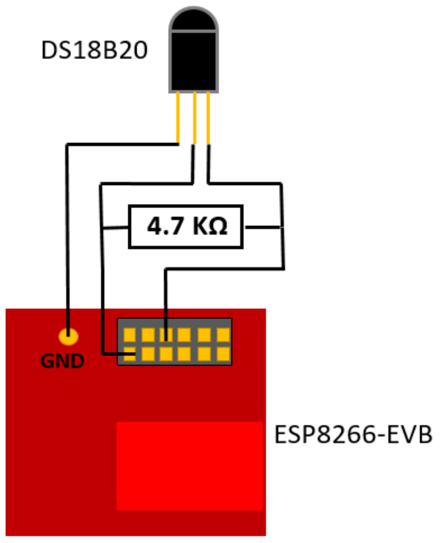


Figure 7.2: Diagram showing how the board was connected to the temperature sensor. Yellow represents the pin of a component, black lines represent wires.

7.5 The Secured Implementation

After the unsecured implementation was up and running, the next point of focus was making it so that the information being sent could not be viewed by a third party. The ability to view this information was tested. The methodology and results of this can be found in section 8.2.2. In short: a third party with access to the traffic could easily read the information in the TCP packets using a tool such as Wireshark or Tshark. This section discusses the implementation of TLS by describing the code relevant to each phase of the protocol. In terms of packet composition, only fields that require some significant computation will be discussed in detail. The "General Fields" section discusses the fields that are common to all packets.

7.5.1 General Methodology

In general, the methods that involve constructing packets were created by looking at real TLS connections. Initially, regular TLS packets were examined. However, towards the end of the project, Stunnel was set up as a client on a different machine, and configured to communicate with the stunnel-enabled server using TLS-PSK. The packets from this interaction were captured, and their composition was emulated (primarily in terms of the relevant extensions). The reasoning behind this was simply to emulate a working implementation of the TLS-PSK handshake.

7.5.2 General Fields

All TLS messages in this implementation are sent in individual records, i.e. there is one message per record, and vice versa. These records all have the content type, TLS version and length fields. Additionally, all messages within these records will have the message type, length, and TLS version fields. These fields will be referred to as "general fields" for the purposes of this chapter. As in chapter 6, fields that come after the aforementioned general fields will be referred to as "inner message fields", for brevity.

7.5.3 Sending Packets

A method was written that would take a packet variable and send it to the server. Packets, in this case, are arrays of type "uint8_t", which is an 8-bit (single byte) unsigned character . The sendpacket() function does this by casting the packet to a const and passing the result, along with the packet's size, to the write() method of an object of the WiFiClient class. This method requires the packet passed to it be a const as a guarantee that it will not be modified by the internal code.

7.5.4 Receiving Packets

Packets sent by the server could be received on one of two ways: calling the client.read() method repeatedly to get all bytes being received, or by using the receivePacket() method. This method takes a vector object and a WiFiClient object as its arguments. It uses the WiFiClient object to read data being sent to the board, and stores it in the supplied vector. The packets are stored for two reasons: firstly so that the packet can be parsed for important information, and secondly so that it can be saved so that a hash of the entire handshake can be generated for the Finished message (see chapter 6).

7.5.5 Pseudo-Random Function

The Pseudo-Random Function is primarily an algorithm that allows a hash of a specific length to be computed. It can be thought of as a secure key expansion algorithm, and works in a manner similar to Cipher Block Chain encryption algorithms. A PRF requires at least 4 arguments: the secret, label, and seed are all used to calculate a new hash at each step. The 4th argument is the desired length of the output. The PRF can be thought of as a function that determines the arguments for, and calls, another function called P_hash. P_hash is outlined below. For brevity, concatenation operations are denoted by the "+" symbol, and "+=" denotes a variable having something else concatenated with it.

Algorithm 1: The P_hash algorithm

```
Input: secret, seed, iterations
Output: result
A = NULL;
for $i < iterations$ do
    A = HMAC(secret, A+seed);
    result += A;
    i++;
end
```

The PRF takes its 4th argument, desired length of output, and calculates the number of iterations of the P_hash required to achieve at least this amount of data. The HMAC function in the in P_hash is, in this implementation, SHA256 HMAC, which returns 32 bytes upon each iteration (since the "256" refers to the number of bits in the output, which is equal to 32 bytes). The secret passed to P_hash is the same secret passed to the PRF, but the seed passed to P_hash is actually the PRF's label+seed. in this implementation, the P_hash algorithm is a part of the PRF method, since it only consists of a single "for" loop with a concatenation and a call to the HMAC method.

7.5.6 Pre-Master Secret

The pre-master secret is a secret value computed by both parties. This value is then used to compute the master secret (the purpose of which is explained in the next subsection). Under TLS-PSK, the pre-master secret is computed as follows:
let N be the length of the PSK, Z be a string of 0's, of length N, and psk be the string representation of the PSK
Pre Master Secret = N + Z + N + psk
The TLS::preMasterSecret() method performs these operations.

7.5.7 Master Secret

The Master Secret is used as a source of entropy when a hash of the handshake messages is being computed. It is computed by both client and server. This is achieved by using the PRF, and passing the pre-master secret as the secret parameter, the string "master secret" as the label, client random concatenated with server random as the seed, and specifying 48 bytes as the desired length of the output. The TLS::masterSecret() method handles these operations.

7.5.8 Client Hello

The Client Hello message packet is constructed using the TLS::client_hello() method. This method puts all the general fields into the supplied packet variable. Following this, client random must be generated. It was found initially that, given the same conditions, the ESP8266 would generate the same "random" data every time it was reset. This is because there is, by default, no good source of entropy. This problem was solved by reading the value of a disconnected pin using the analogRead() function, and using the result in the randomSeed() function. The randomSeed() function is used to set the seed for the random number generator. In this case, the value returned by analogRead() cannot be predicted accurately, so the seed used to generate random numbers can itself be considered to be securely random, and thus, so can the random numbers generated from this. Following client random, the bytes representing supported cipher suites and compression methods are appended to the packet. Finally, some extensions and padding are added to the end of the packet.

7.5.9 Client Key Exchange and Change Cipher Spec

The Client Key Exchange message is reasonably simple to generate: it contains the general fields, and two inner message fields. These are simply the PSK identity length, and the PSK identity. This identity is specified in the sketch that makes use of the library. This message is generated by the TLS::client_key_exchange() method
The Change Cipher Spec message is almost always the same. The only part of it that can change is the TLS version. It contains the general fields, but has no inner message fields. It is generated with the TLS::change_cipher_spec() method

7.5.10 Client Finished

The Client Finished message is the most complicated message to send. The TLS::client_finished() method requires a number of other methods and pieces of information in order to generate the message. The TLS::client_finished() method requires 3 arguments: a vector containing a value called the master secret, a vector containing all handshake messages concatenated together, and an empty array for inserting the finished message into. Initially, the TLS::sha256Hash() method is applied to the vector containing the handshake messages. This method also takes an initialized 32-byte array as a parameter, and copies the 32-byte result of hashing the handshake messages into this empty array. Following this, the PRF is used to generate a 12-byte hash, taking the master secret as its secret parameter, the string "client_finished" as its label, and the previously created 32-byte hash of the handshake messages as the seed. The resulting value is then inserted as the first and only inner message field.

7.5.11 Application Data

Once the handshake is complete, the TLS::applicationData() method is used to create a TLS packet containing encrypted application data. The method takes as arguments an empty array into which to copy the resulting encrypted data, the data itself as a character array, the size of the data, and the PSK to use for encryption. The encryption, in this implementation, is done using an AES-128 CBC algorithm, called Tiny AES128. This implementation requires that the length of the data passed to it is a multiple of 16, since the algorithm operates in blocks of 16. To satisfy this, the data passed to the algorithm is padded with zeroes as necessary to bring it up to the appropriate length. Additionally, AES requires an initialization vector, however this was set to 0 for the purposes of this implementation, and because specific information on generating this value under TLS-PSK could not be found. These are fed as arguments to the method of another class: aes::AES128_CBC_encrypt_buffer(). This encrypts the provided data into a buffer provided. This buffer is then copied into the packet array, and the packet becomes ready to send.

7.6 External Libraries Used

Some external libraries had to be brought in to avoid the need to "roll my own" encryption, which is commonly considered to be a bad idea. Also, considering the number of existing implementations, it would have been unnecessary to reinvent the wheel by using time to create more.

7.6.1 Spaniakos Cryptosuite

A GitHub user whose handle is Spaniakos developed a cryptographic suite designed for Arduinos and Raspberry Pis. It includes implementations of SHA256, and HMAC-SHA256 which were used as part of the PRF in this project.

7.6.2 Tiny AES128

A GitHub user with username "Kokke" developed a C implementation of AES128 in CBC and ECB mode. This implementation was, initially, tested by simply copying the contents of the .h and .c files into a sketch, alongside some included test code that was put into the setup() function. Since it was the only AES implementation tested that could be made to work (Spaniakos has an AES implementation that I was unable to use on the ESP8266), it was decided that it would be modified into a C++ library. This took some time, as well as plenty of trial-and-error, but the library was eventually ported so that it could function as a C++ library, with minimal changes to the code itself. I have no experience of doing so, so it likely that the result is somewhat messy, but it is functional. Additionally, only one function from this library is actually used, and this occurs only once in the TLS library, so it is completely feasible to swap it out for another, more purpose-built implementation.

7.6.3 STL

A port of the Standard Template Library was included so that vectors could be made use of, as they are not included in the somewhat cut-down C++ compiler that ships with the Arduino IDE.

Chapter 8

Testing

The primary aims of testing during and after development were to assess the speed of certain functions, as well as confirming that they produced the correct results. For cryptographic algorithms, the implementations used in this project were checked against implementations found online. These benchmark implementations had been tested (both by the creators and myself) against NIST's provided test data.

8.1 Component Testing

Component testing primarily consisted of testing the algorithms relevant to cryptographic functions. This was done to test both the performance as well as the correctness of the algorithms. Performance was measured in speed, and correctness was given either a "yes it is correct" or "no it is not correct" result. The purpose of testing performance was to ascertain whether the code could be used without slowing down the entire system significantly. The purpose of testing the correctness was simply to ensure that the creators were telling the truth (they were), and that there were no discrepancies caused by using them under the circumstances of this project. Component testing was also relevant to the methods that handle the generating of packets; these all produced the same type of output, so one type of test was sufficient to assess each method.

8.1.1 Testing Tiny AES128

The Tiny AES128 implementation was tested to determine if the speed with which it could encrypt and decrypt data was sufficient for use in the main program. This was achieved by writing a sketch which used the two test functions that were provided with the library, test_encrypt() and test_decrypt(). These functions would encrypt or decrypt some a large amount of test data and record the time, in milliseconds, that it took to do so. They would also print out a "FAILED" statement if the output was not correct. This was checked by including the precalculated output in the method, and using strncmp() to compare this precalculated output with the actual output.
The results of this test are showin in table 8.1

Table 8.1: Average Speeds of AES Encryption and Decryption

	Encryption	Decryption
Average Time Taken (ms)	100	324

Additionally, an AES implementation found online, [28] written in javascript and checked against the NIST test vectors (both by myself and the original developer) was used to verify the

Table 8.2: SHA256 HMAC Performance with NIST Test Vectors

Test Vector	4.2	4.3	4.4	4.5	4.6	4.7	4.8
Time Taken (ms)	1	1	8	9	9	8	8
Success? (Y/N)	Y	Y	Y	Y	Y	Y	Y

correctness of the Tiny AES128 implementation. When encrypting data using the aes implementation, the same data would be encrypted with the javascript implementation in order to verify its correctness.
In terms of results, Tiny AES128 consistently gave the same outputs as the javascript implementation, both for random inputs, and the NIST test vectors.

8.1.2 Testing Cryptosuite

In a similar manner to the Tiny AES128 implementation, the SHA256 Hmac algorithm from Spaniakos' Cryptosuite was tested to determine both its correctness when running on the ESP8266, and its running time. A test script that was provided with the library was modified for use on the ESP8266, and run. Each test run was based on a test vector from RFC 4231 CITATION. The results of running these tests are displayed in table 8.2.

Based on these results, it was determined that the SHA256 HMAC implementation was fast enough, and correct when being run on the ESP8266 board.

8.1.3 Testing the Pseudo-Random Function

The Pseudo-Random Function was tested by calling the function 100 times, measuring the time taken (in ms) each time, and then taking the average time from this. A script was written which would carry this out. It would make the PRF return 1000 bytes of data; far more than would ever be needed. The test was run 10 times, and each time, the average time to expand 1000 bytes of data was 31 milliseconds. This performance was deemed to be acceptable.

8.1.4 Testing Packet Generating Methods

In order to test the methods which generated packets, these packets would simply be sent to the server, and captured and saved to a .pcap file with Tshark. Following this, WinSCP would be used to access the files on the Raspberry Pi, and Wireshark would be used to parse the .pcap file and display the structure of each packet. An example of Wireshark being used to do so can be seen in Figure 8.1.

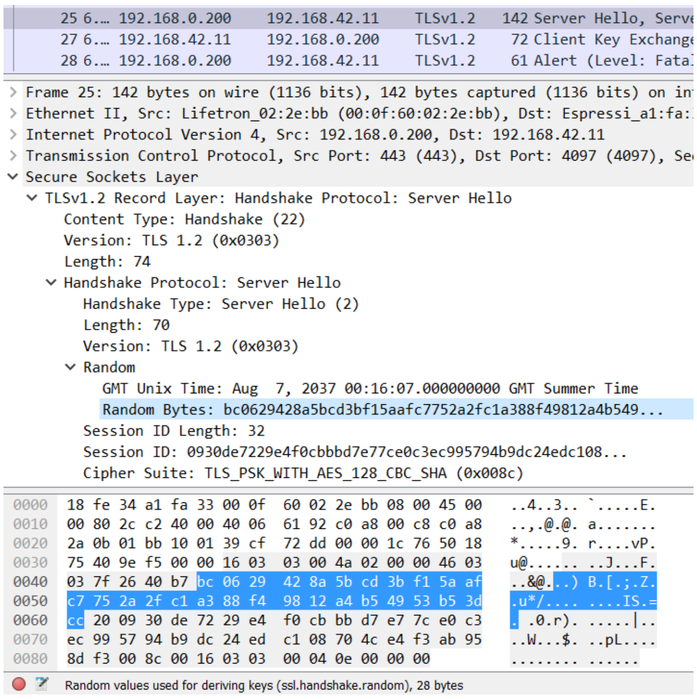
Examining packets in this way made the task of figuring out if any fields were being set incorrectly much, much simpler. This technique was used for testing all packet-related methods.

8.2 Overall System Testing

The purpose of overall system testing (effectively black-box testing) was to determine how reliably the system functions, discover bugs, and ascertain the performance of the system. It was assessed in terms of its speed - namely how quickly a TLS handshake could be performed, as well as how quickly data could be encrypted and sent to the server.

8.2.1 Testing the Unsecured Implementation

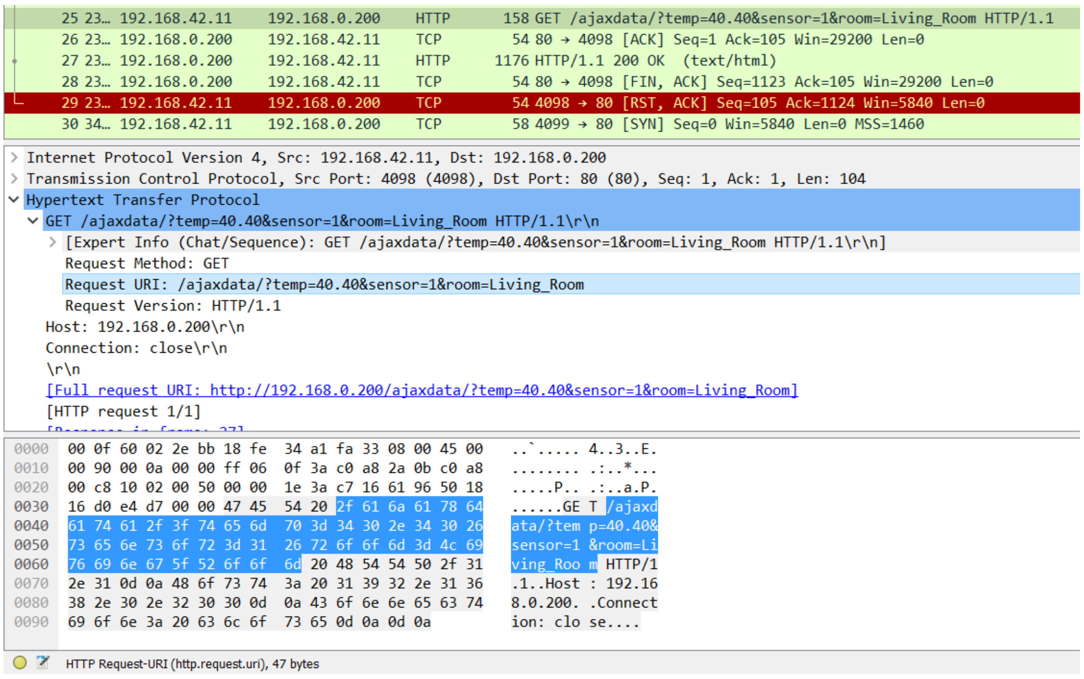
The unsecured implementation contains code that times the execution time for sending data. It should be noted that the sketch features 10 seconds worth of delays between each reading, so



```
25 6… 192.168.0.200    192.168.42.11   TLSv1.2   142 Server Hello, Serve
27 6… 192.168.42.11    192.168.0.200   TLSv1.2    72 Client Key Exchange
28 6… 192.168.0.200    192.168.42.11   TLSv1.2    61 Alert (Level: Fatal

> Frame 25: 142 bytes on wire (1136 bits), 142 bytes captured (1136 bits) on in
> Ethernet II, Src: Lifetron_02:2e:bb (00:0f:60:02:2e:bb), Dst: Espressi_a1:fa:
> Internet Protocol Version 4, Src: 192.168.0.200, Dst: 192.168.42.11
> Transmission Control Protocol, Src Port: 443 (443), Dst Port: 4097 (4097), Se
v Secure Sockets Layer
  v TLSv1.2 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 74
    v Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 70
        Version: TLS 1.2 (0x0303)
      v Random
          GMT Unix Time: Aug  7, 2037 00:16:07.000000000 GMT Summer Time
          Random Bytes: bc0629428a5bcd3bf15aafc7752a2fc1a388f49812a4b549...
        Session ID Length: 32
        Session ID: 0930de7229e4f0cbbbd7e77ce0c3ec995794b9dc24edc108...
        Cipher Suite: TLS_PSK_WITH_AES_128_CBC_SHA (0x008c)

0000  18 fe 34 a1 fa 33 00 0f  60 02 2e bb 08 00 45 00   ..4..3.. `.....E.
0010  00 80 2c c2 40 00 40 06  61 92 c0 a8 00 c8 c0 a8   ..,.@.@. a.......
0020  2a 0b 01 bb 10 01 39 cf  72 dd 00 00 1c 76 50 18   *.....9. r....vP.
0030  75 40 9e f5 00 00 16 03  03 00 4a 02 00 00 46 03   u@...... ..J...F.
0040  03 7f 26 40 b7 bc 06 29  42 8a 5b cd 3b f1 5a af   ..&@...) B.[.;.Z.
0050  c7 75 2a 2f c1 a3 88 f4  98 12 a4 b5 49 53 b5 3d   .u*/.... ....IS.=
0060  cc 20 09 30 de 72 29 e4  f0 cb bb d7 e7 7c e0 c3   . .0.r). .....|..
0070  ec 99 57 94 b9 dc 24 ed  c1 08 70 4c e4 f3 ab 95   ..W...$. ..pL....
0080  8d f3 00 8c 00 16 03 03  00 04 0e 00 00 00         ........ ......

Random values used for deriving keys (ssl.handshake.random), 28 bytes
```

Figure 8.1: Snapshot of the Wireshark Interface

10 seconds of each reading can be attributed to this. When running the program for a number of minutes, the average time taken to send a packet to the Raspberry Pi server, and receive a response, was 11,163.67 ms. When adjusting this for the delays, the time spent sending and receiving data from the server was, on average, 1,163ms, or approximately a second.
Additionally, Tshark was used to capture packets when the implementation was running, and Wireshark was used to examine them. The result is displayed in figure 8.2.



```
25 23… 192.168.42.11   192.168.0.200   HTTP    158 GET /ajaxdata/?temp=40.40&sensor=1&room=Living_Room HTTP/1.1
26 23… 192.168.0.200   192.168.42.11   TCP      54 80 → 4098 [ACK] Seq=1 Ack=105 Win=29200 Len=0
27 23… 192.168.0.200   192.168.42.11   HTTP   1176 HTTP/1.1 200 OK  (text/html)
28 23… 192.168.0.200   192.168.42.11   TCP      54 80 → 4098 [FIN, ACK] Seq=1123 Ack=105 Win=29200 Len=0
29 23… 192.168.42.11   192.168.0.200   TCP      54 4098 → 80 [RST, ACK] Seq=105 Ack=1124 Win=5840 Len=0
30 34… 192.168.42.11   192.168.0.200   TCP      58 4099 → 80 [SYN] Seq=0 Win=5840 Len=0 MSS=1460

> Internet Protocol Version 4, Src: 192.168.42.11, Dst: 192.168.0.200
> Transmission Control Protocol, Src Port: 4098 (4098), Dst Port: 80 (80), Seq: 1, Ack: 1, Len: 104
v Hypertext Transfer Protocol
  v GET /ajaxdata/?temp=40.40&sensor=1&room=Living_Room HTTP/1.1\r\n
    > [Expert Info (Chat/Sequence): GET /ajaxdata/?temp=40.40&sensor=1&room=Living_Room HTTP/1.1\r\n]
      Request Method: GET
      Request URI: /ajaxdata/?temp=40.40&sensor=1&room=Living_Room
      Request Version: HTTP/1.1
    Host: 192.168.0.200\r\n
    Connection: close\r\n
    \r\n
    [Full request URI: http://192.168.0.200/ajaxdata/?temp=40.40&sensor=1&room=Living_Room]
    [HTTP request 1/1]

0000  00 0f 60 02 2e bb 18 fe  34 a1 fa 33 08 00 45 00   ..`..... 4..3..E.
0010  00 90 00 0a 00 00 ff 06  0f 3a c0 a8 2a 0b c0 a8   ........ .:..*...
0020  00 c8 10 02 00 50 00 00  1e 3a c7 16 61 96 50 18   .....P.. .:..a.P.
0030  16 d0 e4 d7 00 00 47 45  54 20 2f 61 6a 61 78 64   ......GE T /ajaxd
0040  61 74 61 2f 3f 74 65 6d  70 3d 34 30 2e 34 30 26   ata/?tem p=40.40&
0050  73 65 6e 73 6f 72 3d 31  26 72 6f 6f 6d 3d 4c 69   sensor=1 &room=Li
0060  76 69 6e 67 5f 52 6f 6f  6d 20 48 54 54 50 2f 31   ving_Roo m HTTP/1
0070  2e 31 0d 0a 48 6f 73 74  3a 20 31 39 32 2e 31 36   .1..Host : 192.16
0080  38 2e 30 2e 32 30 30 0d  0a 43 6f 6e 6e 65 63 74   8.0.200. .Connect
0090  69 6f 6e 3a 20 63 6c 6f  73 65 0d 0a 0d 0a         ion: clo se....

HTTP Request-URI (http.request.uri), 47 bytes
```

Figure 8.2: Snapshot of the Wireshark Interface

Table 8.3: Comparing the Secured and Unsecured Implementations

Action:	TLS-PSK Handshake	Sending Encrypted TLS Application Data	Sending Unencrypted Data
Time (ms)	4221	76	6

The GET request being send from the ESP8266 can clearly be seen in the packet, as it is being sent as plaintext. This is how it would appear to anyone who could intercept the packets, and demonstrates the need for secure communications.

8.2.2 Testing the Secured Implementation

A sketch was written which would perform a TLS-PSK handshake and then send application data in two manners: unencrypted (as in the unsecured implementation) and encrypted and in a TLS packet. The handshake and both attempts to send application data were timed. The experiment would send 100 lots of data, and take the average time, in milliseconds, that was taken in order to send the data. The results are displayed in table 8.3.

From these results, it can be seen that the secured implementation, while significantly slower than the secured one, is sufficiently fast that multiple readings can be encrypted and transmitted per second. Therefore, using AES128 with an ESP8266 is completely feasible for real-time applications that require anything up to 10 small transmissions per second. It was noticed that the average time to perform the encryption in this scenarion was markedly less than that of previous AES128 testing. This was because the data being encrypted here was a simple GET request, which consisted of significantly fewer characters than the test data. The isolated AES test data was deliberately larger than the data used in this test, because it is possible that users might want to send more information than was sent in this scenario.

8.3 Known Issues

There are some reliability issues with the software: when uploading the code for the first time, it is very likely that the handshake will cause the ESP8266 to crash when it tries to send the first message. However, simply restarting the board after uploading the code fixes this problem. Additionally, the server that was being used to communicate with would continually respond to with handshake errors, despite the handshake messages being specifically designed to look identical to the messages sent between stunnel clients. This may be attributable to the fact that the TCP connection between the client and server is being dropped. To prevent this, an implementation that sends the packets using C sockets or similar might be preferable. This could be implemented by simply changing the sendPacket() method in the TLS code.

Chapter 9

Cost-Benefit Analysis

This section compares the ESP8266 to some other available platforms. Each platform is assessed in a number of ways, namely:

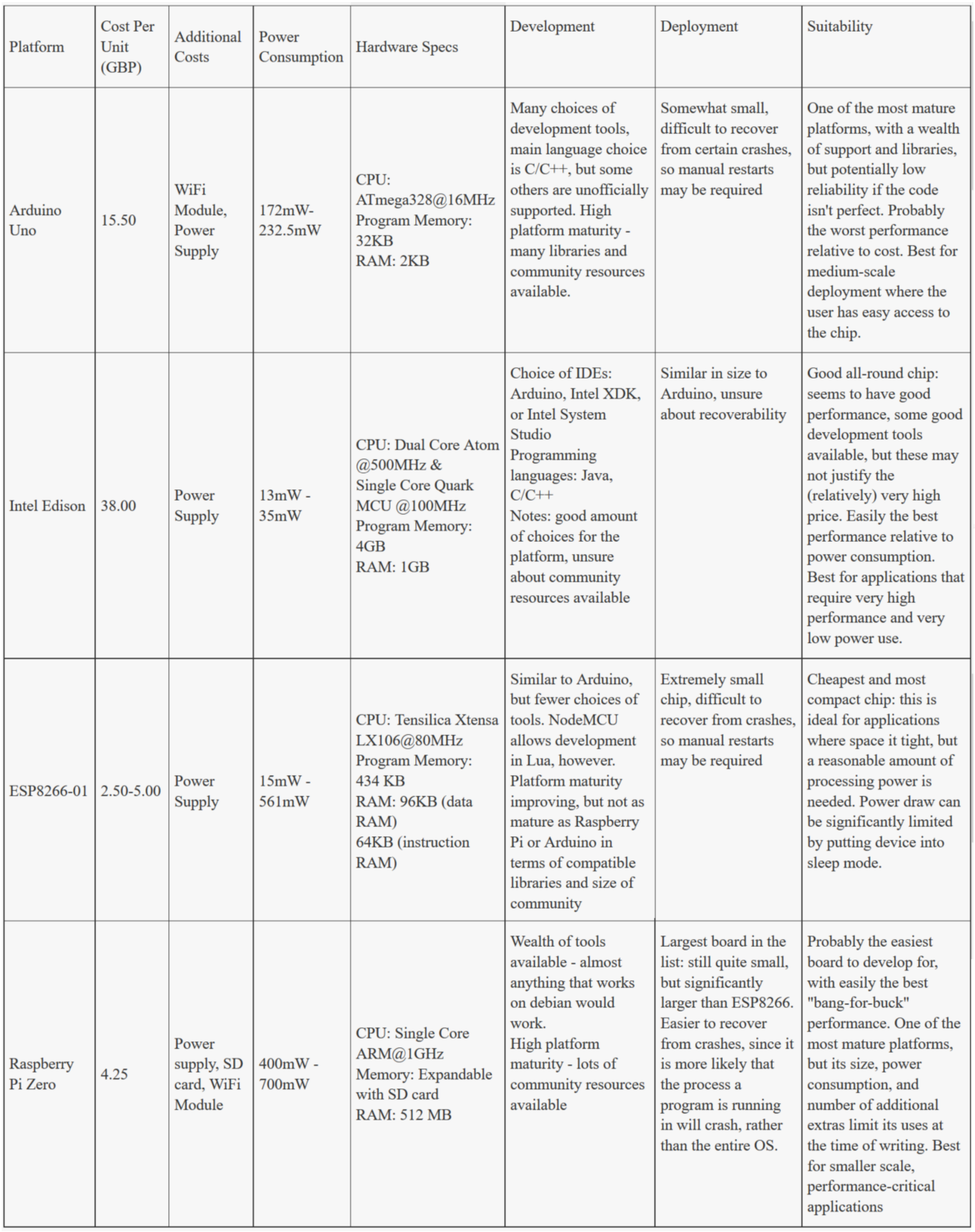
- cost - price per unit
- additional costs - cost of additional components
- power consumption - in kW/H
- hardware specifications - how much performance can you expect?
- development - factors that affect the development process of the platform
- deployment - factors affecting how easy it is to deploy initially, and maintain after the fact

Figure 9.1 displays some important information about a number of competing boards. the final column in the table - Suitability - outlines my estimate of which types of projects would be more, or less, appropriate for the platform. Information about the boards was primarily sourced from the manufacturer's pages. Where there was information missing from these, mouser.com was used.[1] Additionally, some of the power estimates were taken from investigations found online [2][3][4]

9.1 Analysis

This section primarily consists of a table that compares 4 IoT boards. Each board was researched, and specifications were generally taken from the manufacturer's website, where possible.

[1]http://www.mouser.co.uk
[2]http://www.jeffgeerling.com/blogs/jeff-geerling/raspberry-pi-zero-power
[3]http://gadgetmakersblog.com/arduino-power-consumption/
[4]http://bbs.espressif.com/viewtopic.php?t=133



Platform	Cost Per Unit (GBP)	Additional Costs	Power Consumption	Hardware Specs	Development	Deployment	Suitability
Arduino Uno	15.50	WiFi Module, Power Supply	172mW-232.5mW	CPU: ATmega328@16MHz Program Memory: 32KB RAM: 2KB	Many choices of development tools, main language choice is C/C++, but some others are unofficially supported. High platform maturity - many libraries and community resources available.	Somewhat small, difficult to recover from certain crashes, so manual restarts may be required	One of the most mature platforms, with a wealth of support and libraries, but potentially low reliability if the code isn't perfect. Probably the worst performance relative to cost. Best for medium-scale deployment where the user has easy access to the chip.
Intel Edison	38.00	Power Supply	13mW - 35mW	CPU: Dual Core Atom @500MHz & Single Core Quark MCU @100MHz Program Memory: 4GB RAM: 1GB	Choice of IDEs: Arduino, Intel XDK, or Intel System Studio Programming languages: Java, C/C++ Notes: good amount of choices for the platform, unsure about community resources available	Similar in size to Arduino, unsure about recoverability	Good all-round chip: seems to have good performance, some good development tools available, but these may not justify the (relatively) very high price. Easily the best performance relative to power consumption. Best for applications that require very high performance and very low power use.
ESP8266-01	2.50-5.00	Power Supply	15mW - 561mW	CPU: Tensilica Xtensa LX106@80MHz Program Memory: 434 KB RAM: 96KB (data RAM) 64KB (instruction RAM)	Similar to Arduino, but fewer choices of tools. NodeMCU allows development in Lua, however. Platform maturity improving, but not as mature as Raspberry Pi or Arduino in terms of compatible libraries and size of community	Extremely small chip, difficult to recover from crashes, so manual restarts may be required	Cheapest and most compact chip: this is ideal for applications where space it tight, but a reasonable amount of processing power is needed. Power draw can be significantly limited by putting device into sleep mode.
Raspberry Pi Zero	4.25	Power supply, SD card, WiFi Module	400mW - 700mW	CPU: Single Core ARM@1GHz Memory: Expandable with SD card RAM: 512 MB	Wealth of tools available - almost anything that works on debian would work. High platform maturity - lots of community resources available	Largest board in the list: still quite small, but significantly larger than ESP8266. Easier to recover from crashes, since it is more likely that the process a program is running in will crash, rather than the entire OS.	Probably the easiest board to develop for, with easily the best "bang-for-buck" performance. One of the most mature platforms, but its size, power consumption, and number of additional extras limit its uses at the time of writing. Best for smaller scale, performance-critical applications

Figure 9.1: Cost-Benefit Analysis of 4 competing IoT Boards

9.2 Findings

This section discusses inferences that can be made from Figure 9.1. Each board is discussed in terms of its strengths and weaknesses, and how these affect the type of projects that the board is most suited to. It should be noted that, when a type of project is not discussed for a board, this does not mean that it would not perform well in this context, it simply means that the type of project that was discussed appears more well-suited.

9.2.1 Arduino Uno

From the table, it can be seen that each board has its own advantages. The Arduino would likely be most well-suited to small-scale or DIY projects, due to the fact that the platform is significantly more mature than most other boards. It could also function well in larger scale projects, but its price relative to its performance mean that other boards would be preferable in areas that require higher performance on a low budget.

9.2.2 Intel Edison

The Intel Edison board would be ideal where low power consumption and fairly high performance is critical, however these factors would need to be extremely important to justify its price, which is approximately 3-10 times that of the othe boards listed here.

9.2.3 Raspberry Pi Zero

The Raspberry Pi would be, at present, ideal for a wide range of projects. However, it may not be ideal for industrial deployment because its high power consumption, relative to the other boards, could significantly limit the amount of time that the board could operate when powered by a battery. Additionally the requirement for extra components such as a WiFi module and micro SD card would have an impact on costs.

9.2.4 ESP8266

Finally, the ESP8266 appears to be quite well suited to more large-scale deployments. Its small size makes it easier to hide and removes some of the burden of finding space and housing the board. Its low cost means that using these boards in mass-production is completely feasible, and its surprisingly high performance at this price broadens the scope of applications that could be developed for this board. At present, the main issue with this board is the immaturity of its environment, but this is likely to change with time.

Chapter 10

Conclusion and Discussion

This chapter concludes the project. Findings based on the work done are summarized in section 10.1, and the implications of these findings are discussed in section 10.2. Suggestions for future work, as well as the most significant challenges and shortcomings of this project, are given in the section 10.2.3.

10.1 Conclusions

The functional requirements outlined in section4.1 have largely been met. Chapter 8 addressed requirements 4.1.1 and 4.1.3, where the unsecured connection was compared to a secured one. It was found that, once the handshake has completed, sending encrypted data took only an average of 76ms, which is acceptable for the majority of applications where the frequency of data being sent is anything less than around 10 readings per second. Additionally, requirement 4.1.2 was satisfiet to an extent. While the majority of features required for a successful TLS-PSK connection are implemented, they could not be made to work with stunnel. This is either because stunnel server processes may be primarily designed to work with other stunnen-enabled clients, or because of the nature of the WiFiClient class from the ESP8266 library, which may have been trying to create new TCP connections too frequently in this implementation. Finally, requirement 4.1.4 was inherently satisfied by the development process; it would have been very difficult to leave this requirement unsatisfied, as looking at the actual composition of packets from a real TLS/TLS-PSK handshake was instrumental in informing how to compose them in this implementation.

The non-functional requirements were each satisfied throughout the development process: the TLS library turned out to be a single file, from which the user need only know two commands, (requirement 4.2.1), the relevant RFCs were consistently consulted to inform design (requirement 4.2.2), and only open-source external software was used to suppliment the implementation (requirement 4.2.3).
The main product of this work, the TLS-PSK implementation, was primarily a proof of concept. However, the code is written in such a way that it is completely feasible to build upon it, and even expand it into an equally easy-to-use TLS implementation involving other, potentially non-PSK cipher suites.

10.2 Discussion

10.2.1 Evaluation of the ESP8266

As seen in chapter 9, the ESP8266 is an extremely cheap WiFi chip that has the potential to be deployed as part of large-scale IoT projects, as well as small home-automation-type projects. It benefits from having fairly good performance for its price, as well as being surprisingly compact. Its power consumption appears to be fair, meaning that it could, if programmed correctly, operate for a long time when powered by a battery. However, there are some challenges that come from developing on this platform. While the community is sizeable and responsive, the platform has not had as much time as others, such as the Arduino or Raspberry Pi, to mature. There are still not many unofficial libraries purpose-built for the ESP8266 and modifying existing libraries to make them compatible was required during this project.

10.2.2 The TLS-PSK implementation

As a proof of concept, it has proven that implementing TLS-PSK on a small chip like this is feasible, but the fact that it couldn't be made to work with stunnel suggests that developing strongly networking-related applications for this platform is a challenging. The individual components of the implementation appeared to work well, however the testing of messages beyond the client key exchange was less vigorous, because the server would consistently reject the handshake, even if the messages appeared exactly the same as those of a successful handshake.

As stated in section 10.1, this implementation sets a baseline from which other implementations can be made: the code is set out in such a way that it should be easy to add new methods, or edit existing ones. Had there been even more time for this project, matching server-side code could have been written to function with the client-side TLS-PSK implementation. Additionally, the WiFiClient class would likely have been moved away from in favour of using code that allows more fine-grained control over the TCP connections made from the board.

10.2.3 Shortfalls and Future Improvements

The main issues with this project were that the TLS-PSK implementation could not be made to work with the server. This was in part caused by the fact that TLS-PSK is not a commonly used cipher suite, and this is not supported by many server applications, e.g. the well-known Apache, which supports TLS, but not PSK cipher suites. Additionally, there were some issues with reliability of the system: when the first upload of code is made, the system commonly fails when it tries to send the first TLS packet. However, restarting the system after code upload is complete remedies this issue.

In future, work would be done to improve the reliability of the application, and to compare similar implementations on other devices. In addition to this, the TLS-PSK implementation could be improved so that it allows a more broad range of cipher suites. To supplement this, it would also be advantageous for more cryptographic algorithms to have implementations ported to the platform, such as Diffie-Helmann and elliptic curve algorithms.

Bibliography

[1] "Arduino uno r3 (atmega328 - assembled)," 2016. Available at: https://www.adafruit.com/products/50.

[2] "Esp8266 wifi module." Available at: https://www.adafruit.com/products/2822.

[3] "Arduino due - assembled - due," 2016. Available at: https://www.adafruit.com/products/1076.

[4] P. Eronen and H. Tschofenig, "Rfc4279: Pre-shared key ciphersuites for transport layer security (tls)," tech. rep., 2005.

[5] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of things," *International Journal of Communication Systems*, vol. 25, no. 9, p. 1101, 2012.

[6] C. Buckley, "ChinaâĂŹs vaccine scandal threatens public faith in immunizations," *New York Times*, 2013.

[7] "Support for amazon dash button," 2015. Available at: http://www.amazon.com/gp/help/customer/display.html/ref=hp_bc_nav?ie=UTF8\&nodeId=201706050.

[8] "Does it have an official programming language?," 2016. Available at: https://www.raspberrypi.org/help/faqs/\#softwareLanguages.

[9] "Nodemcu, connect things easy," 2016. Available at: http://nodemcu.com/index_en.html.

[10] B. Benchoff, "New chip alert: The esp8266 wifi module (itâĂŹs $5)," *hackaday.com*, 2014.

[11] M. Higashi, "Symmetric vs. asymmetric encryption - which is best?," 2013.

[12] B. Lovejoy, "Chromecast vulnerability to hijacking demonstrated by rick-roll," 2014. Available at: http://9to5google.com/2014/07/21/chromecast-vulnerability-to-hijacking-demonstrated-by-rickroll.

[13] B. Ostermaier, M. Kovatsch, and S. Santini, "Connecting things to the web using programmable low-power wifi modules," in *Proceedings of the Second International Workshop on Web of Things*, p. 2, ACM, 2011.

[14] T. Heer, O. Garcia-Morchon, R. Hummen, S. L. Keoh, S. S. Kumar, and K. Wehrle, "Security challenges in the ip-based internet of things," *Wireless Personal Communications*, vol. 61, no. 3, pp. 527–542, 2011.

[15] S. Tozlu, M. Senel, W. Mao, and A. Keshavarzian, "Wi-fi enabled sensors for internet of things: A practical approach," *Communications Magazine, IEEE*, vol. 50, no. 6, pp. 134–143, 2012.

[16] M. Rudinskiy, "Ece 4999 independent project: Wi-fi communication using esp8266 & pic32 mikhail rudinskiy completed for dr. bruce land," 2014.

[17] O. W. A. S. Project, "Owasp top 10 internet of things vulnerability categories," 2014.

[18] M. Sethi, J. Arkko, and A. Keranen, "End-to-end security for sleepy smart object networks," in *Local Computer Networks Workshops (LCN Workshops), 2012 IEEE 37th Conference on*, pp. 964–972, IEEE, 2012.

[19] A. Ardiri, "Feasibility of security in micro-controllers," 2014.

[20] J.-P. Kaps, *Cryptography for ultra-low power devices*. PhD thesis, WORCESTER POLYTECHNIC INSTITUTE, 2006.

[21] "mbed tls v2.2.1 source code documentation," 2015. Available at: `https://tls.mbed.org/api`.

[22] T. Dierks and E. Rescorla, "The transport layer security (tls) protocol version 1.2, august 2008. url http://www. ietf. org/rfc/rfc5246. txt," tech. rep.

[23] "Httpd - apache2 web server," 2016. Available at: `https://help.ubuntu.com/lts/serverguide/httpd.html`.

[24] A. Brown, "The standart template library (stl) for avr with c++ streams." Available at: `http://andybrown.me.uk/2011/01/15/the-standard-template-library-stl-for-avr-with-c-streams/`, year = 2011.

[25] "Programming esp8266-evb with arduini ide." Available at: `https://olimex.wordpress.com/2015/03/31/programming-esp8266-evb-with-arduino-ide/`, year=2015.

[26] "setup()," 2016. Available at: `https://www.arduino.cc/en/Reference/Setup`.

[27] "loop()," 2016. Available at: `https://www.arduino.cc/en/Reference/Loop`.

[28] "Tiny aes128 in c," 2015. Published by user: Kokke. Available at: `https://github.com/kokke/tiny-AES128-C`.

Appendix A

Appendix

A.1 User Manual

This guide will describe how to set up an ESP8266-EVB so that it can read the temperature from a DS18B20 sensor, and send it to a remote server.

A.1.1 Prerequisites

This subsection assumes that you do not have a server to connect to, and need to set one up. The required hardware components to communicate with an existing server are:

- ESP8266-EVB board
- 5V DC power supply
- USB to serial converter
- DS18B20 temperature sensor
- female-female jumper cables x 3
- 1x male-male jumper cable
- 2x female-male jumper cables
- breadboard

In order to replicate the test bed in this project, some additional hardware is neede:

- Raspberry Pi (any model with at least 1x USB port and 1x Ethernet port)
- power supply
- Ethernet cable
- compatible USB WiFi module

Finally, Arduino IDE[1] is required to upload the sketches.

[1] `https://www.arduino.cc/en/Main/Software`

A.1.2 Configuring Arduino IDE

First, the Arduino IDE [2] must be installed on your system. Simply download and run it to get started. Next, ESP8266 compatability needs to be added. To do this, follow the steps below:

1. open the IDE, then click File>Preferences
2. in the "Additional Board Manager URLs" section, paste the following address and then click "ok":
   `http://arduino.esp8266.com/stable/package_esp8266com_index.json`
3. next, navigate to Tools>Boards>Boards Manager. Find the "esp8266" board and click "Install"
4. finally, click Tools>Board>Generic ESP8266 Module

Once this is done, connect the ESP8266-EVB to your computer using your USB to serial converter, and three female-female jumper cables. Connect ground (gnd) on the USB connecter to UEXT pin 2 (numbers are printed on the underside of the board), then TXD of the converter to UEXT pin 4, and finally RXD of the connector to UEXT pin 5.
In order to upload code, first begin by loading up an example sketch in the IDE. After this, hold down the white button on the board (it's quite large and hard to miss), and turn on the power supply to the board while holding down this button. Keep it pressed for a couple of seconds after switching on the power. This puts the board into bootloader mode, which allows you to overwrite the existing firmware with your own C++ programs. Repeat the process of switching the board off, holding the button, and switching it on every time you wish to reupload code.

A.1.3 Installing Libraries

A number of different libraries are required to get the board working. Most of what is needed can be found in a Git repository I created [3]. This download includes the TLS-PSK library, as well as the modified AES128 and SHA256 libraries. Simply open copy the "TLS", "TinyAES", and "SHA" folders into your Arduino libraries folder. This can be found at "C:/Users/%USERNAME%/Documents/Arduino/libraries. Simply include "<TLS.h>" in your sketch to use it.
Before attempting to use the library, one final thing must be done: the port of STL[4] (Standard Template Library) must be installed. Simply download it from the link provided in the footnotes and copy all the files from "avr-stl/include" into your "hardware/tools/avr/avr/include" directory. The "hardware" folder is found in the install directory for Arduino IDE, not the documents folder where your libraries are located.

A.1.4 Configuring your Sketch

modify sketch lines for: server address, ssid, password, port (implementation dependent) In order to use the TLS-PSK library, the sketch must have some constant values defined. Const-qualified character arrays must be defined for the address of the server to be communicated with, the SSID

[2] `https://www.arduino.cc/en/Main/Software`
[3] `https://github.com/AHunter5122/TLS-PSK-required-libs/tree/master`
[4] `http://andybrown.me.uk/2011/01/15/the-standard-template-library-stl-for-avr-with-c-streams/`

of the access point, the password for the access point, the pre-shared key of the server, and the PSK identifier associated with this key. A const-qualified integer is also required to represent the port you are communicating with (usually 443). Finally, a WiFiClient object must be instantiated. The following code snippet shows how to perform a TLS handshake, and then send data to the server:

```
TLS::handshake(psk, psk_identifier, client, server_address, port);
//get (non-const) char array of data here..
TLS::sendAppData(data, length_of_data, client, psk, server_address, port);
```

A.2 Maintainence Manual

This section discusses how to analyze the packets being sent by the system, so as to pinpoint sources of error when debugging.

A.2.1 Prerequisite Software

The software elements required for packet inspection are outlined below:

- (Only required for Windows) SSH software, e.g. PuTTY[5]
- SCP client, e.g. WinSCP[6]
- (optional) Wireshark[7]

A.2.2 Raspberry Pi Configuration

This section describes setting up the Raspberry Pi as an access point and using it for packet capture. It can also be used as a server, but because this is such a common use for a Raspberry Pi, this will not be covered here. (use [8], [9], and [10])
In order to set up the Rasoberry Pi as an access point for packet sniffing, use the Hostapd package. Installation instructions can be found on the Adafruit website [11].
Once this is complete, install the Tshark package using apt-get. After installing Tshark, you must add your username to the wireshark group. Run the following command:

```
# sudo usermod -a -G wireshark <username>
```

where <username> is your user name. After doing this, you must log out and log back in for the changes to take effect, else tshark will refuse to run.
At this point, it should be possible to connect your ESP8266 to the wireless interface of the Raspberry Pi and capture packets on it. Information regarding packet capture can be found on the tshark manpage. [12]
An example command for capturing 100 packets from the wireless interface to a file called "capture.pcap" in the current directory would be:

[5] http://www.putty.org
[6] https://winscp.net/eng/index.php
[7] https://www.wireshark.org
[8] https://www.raspberrypi.org/documentation/remote-access/web-server/apache.md
[9] https://www.digitalocean.com/community/tutorials/how-to-set-up-an-ssl-tunnel-using-stunnel-on-ubuntu
[10] https://www.stunnel.org/auth.html
[11] https://learn.adafruit.com/setting-up-a-raspberry-pi-as-a-wifi-access-point/install-software
[12] https://www.wireshark.org/docs/man-pages/tshark.html

```
# tshark -w capture.pcap -i wlan0 -c 100
```

A.2.3 Analyzing the Packets

In order to analyze the packets, they needed to be retrieved from the Raspberry Pi so that they could be displayed in Wireshark. For this, the following pieces of software were required:

Once these are all set up, they can be used to communicate with the Raspberry Pi. Use WinSCP to access the files on the Raspberry Pi, find the .pcap file with the packets, and simply open it with Wireshark. This will provide a useful GUI for inspecting packets to find any errors.