UNIVERSITY OF STRATHCLYDE

TECHNICAL REPORT

Secure Internet Of Things Sensor Platform

Author:
Adam Kidd

Supervisor:
Dr. James Irvine

Examiner:
Dr Alex Coddington

Phd Student:
Grieg Paul

A technical report submitted in fulfillment of the requirements for the degree of Computer and Electronic Systems

in the

Department of Electronic & Electrical Engineering

March 29, 2016

Declaration of Authorship

I, Adam Kidd, "I hereby declare that this work titled, "Secure Internet Of Things Sensor Platform" has not been submitted for any other degree/-course at this University or any other institution and that, except where reference is made to the work of other authors, the material presented is original and entirely the result of my own work at the University of Strathclyde under the supervision of .. insert supervisor's name."

Signed: _____

Date: _____

UNIVERSITY OF STRATHCLYDE

Abstract

Faculty Of Science
Department of Electronic & Electrical Engineering

Computer and Electronic Systems

Secure Internet Of Things Sensor Platform

by Adam Kidd

This report aims to explain how to build a sensor platform that can transmit data across the internet securely and show that security on low powered microcontrollers is possible and easy. It introduces a light weight cryptographic library and demonstrates it's effectiveness on a low powered system and provides considerations on the level of security it provides.

Contents

List of Figures

Chapter 1

Introduction

The Internet of Things or IoT is the concept of a huge network of physical objects connected and communicating between themselves and to the world wide web. Devices can include domestic appliances, cars and even buildings. It is a rapidly growing field with over 50 million devices expected to be connected to the web by 2020[1]. As such the security of the transmissions of these devices is becoming a more and more pressing issue. IoT's main benefits are the remote control of devices and appliances, the ability of the device to send information about it's state, such as a vending machine reporting that it has run out of a certain item, and to allow the machines to be more automated and to work with other machines, like a home hub device that can turn on the lights and central heating when an occupant is arriving home, with the lights and heating not being connected to each other but to the central hub.

However IoT benefits will be severely limited if it unsecure. IoT is an emerging field but there have already been some high profile security disasters. Ranging from relatively less serious problems such as some attackers being able to glean WiFi network information from internet connected lights[2] to the very concerning and potentially fatal security breaches like someone gaining unauthorised access to your car and assuming control. There have been three examples of this in the last few years with a Jeep Cherokee[3], a Toyota Prius[4] and a Tesla Model S[5] being the cars effected. The hackers were able to control the accelerator, door locks and brakes, among other things. In the Jeep Cherokee the attackers remotely disabled the engine of the car as it was driving up a freeway at 70 MPH. This highlights a very real problem that will only become more important. Too often security is an afterthought but it really needs to be built into products from the offset.

The challenge is to provide a cryptographic solution that is similar in strength of security to solutions that are implemented on more powerful servers and computers but on much smaller and less powerful devices. A solution that performs at an acceptable speed, good security with reduced power consumption.

Google and British Gas have recently released Nest and Hive respectively. Nest was released in 2014 and Hive in 2013. These both involve controlling your central heating remotely and programming in days when you won't be at home and therefore have no need of heating. However, it was two whole years later upon independent investigators discovering that information, about dates when the heating was on because the occupants

were in and off when they were away from home, was being sent unencrypted that British Gas Hive decided to encrypt their products transmissions. With the release of Hive 2 they patched the problems found but they should never have been there. Found in the same investigation, Google had a lesser fault with Nest which was sending the post code of the user unencrypted, which has since been patched. It is only when caught or there is a high profile breach that companies take the steps to fully secure their customers information [6].

This is a new implementation of an old problem, developers and companies sometimes don't employ effective security on their private data, moving to smaller internet connected devices doesn't change this fundamental problem. There are studies about and examples of cryptographic systems for microcontrollers, it is not something that cannot be done. However, perhaps it is too difficult at present for smaller development teams or has too much of a foot print in terms of computational resources and time that it gets pushed to the side. Data security is a fundamentally important concept and one that is necessary for the implementation of all applications, especially those that involve user's private data otherwise there is the potential for loss of money, intellectual property, goods, reputation and health. These problems affect both companies and the consumer.

This project will look into creating a solution that can be easily implemented, provides a full library of authenticated encryption with both asymmetric and symmetric encryption, has a small code size and acceptable performance and security. It will then be used to secure the private data of a user as it is sent across unsecure networks. The example used to work with is the secure transmission of a users private temperature data. If they have a system that monitors the temperature of rooms, that data can be used to figure out when they are likely to be home or not. So, using a base platform in the user network that has access to the temperature sensors throughout the house, it takes the sensor data, signs then encrypts it before sending it to a remote server.

Chapter 2

Background

2.1 Cryptography

Cryptography is the practise of and study of techniques for secure communication in the presence of attackers. To do so, one can use encryption where by messages are encoded in such a way that only authorised parties, or at least parties in possession of the keys, can view them. Digital signatures can also be used to provide authentication. There are two main ways of encryption Symmetric Key encryption and Asymmetric Key encryption. In Symmetric Key encryption both parties have the same key which can encrypt and decrypt messages that are sent between them. The problem is that if Bob wants to send an encrypted message to Alice, he must get the secret key to her. Currently the most secure way for the transmission of secret keys is to hand them over in person, in private. This this project Asymmetric Key encryption and Digital signatures.

2.1.1 Asymmetric Key Encryption

To get round the problem of secure sending of secret keys, one can use Asymmetric Key encryption, a visual depiction can be seen in figure 2.1, which involves a secret key and a public key, the secret key is generally random or pseudo random data and is used to generate a corresponding public key which is mathematically linked to the secret but it is computationally infeasible to calculate the secret key back from the public key. This public key can be given out freely and is not a secret. So if Bob sends a message to Alice he encrypts the message with her public key and she can decrypt it with her secret key. Anyone can encrypt with the public key but only the secret key can decrypt. This type of key encryption gets past the sharing secret key problem, as long as you can make sure the public key is indeed the correct one, but it only stops attackers from reading the message and altering it in transit. However the receiver cannot know who sent him the message as encryption does not prove who the message was sent by.

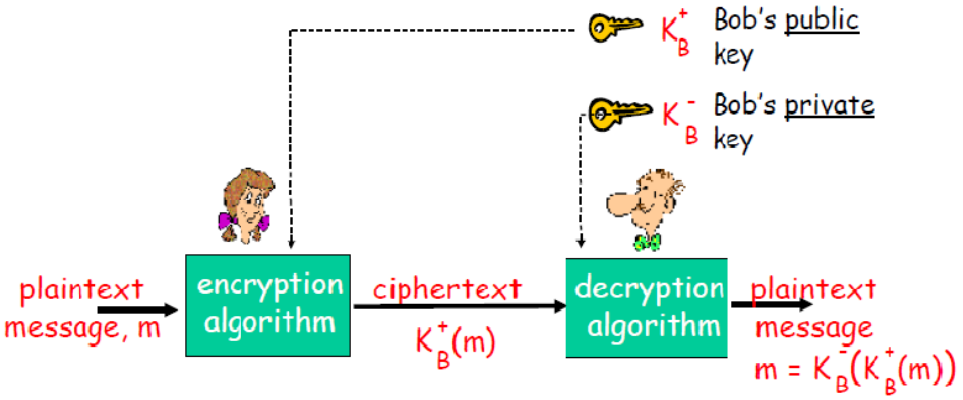

FIGURE 2.1: Asymmetric Key Encryption

2.1.2 Digital Signature

This is a mathematical scheme for proving message authenticity and message non-repudiation. Similarly to asymmetric key encryption a random secret key is created with a corresponding public key. The algorithm takes in a message and a secret key and using SHA-512, it creates a hash using the secret key and message. It then appends that signature to the message. If Alice signs a message in this way, Bob can use another algorithm to verify the message with the public key and signature. It is important to note that the data is not hidden at this point, it is still in plain text.

2.1.3 Nonce

A cryptographic nonce is some arbitrary data that is used in the encryption process and it can only be used once. It will most likely be random or psuedo random data and it is incorporated into cryptographic systems in some way such that old communications that an attacker may have captured can't be sent again and accepted as normal. So the synchronisation of nonces is very important. Some systems use counters as nonces or the nonces on each side are dependent on the same calculation or timing and some systems send the next nonce to be used encrypted.

2.1.4 Message Authentication Code

A MAC is a small piece of information that is used to authenticate a message and prove it's integrity. The MAC algorithm takes in a message and a key and produces a MAC, then it is sent with the message and the receiver runs the algorithm like the sender, with the same message and key, and if the resulting MAC is the same as the received MAC then the message has not been altered and proves the by a person in possession of the same key. MACs differs to digital signatures in that it does not provide message repudiation.

2.1.5 Stream Cipher

A stream cipher is a symmetric key cipher where the plain text message is combined with a cipher digit stream so that each digit of the data is encrypted individually. The combining operation is an XOR. This can be useful for applications that are constantly sending data, for example streaming encrypted video.

2.2 TweetNaCl

NaCl or "Salt" is a simple to use high-speed library for authenticated encryption. It provides both Asymmetric and Symmetric encryption, authentication and message integrity with SHA-512. The authors are Daniel J. Bernstein, Tanja Lange and Peter Schwabe but at points it does rely on some third part implementations. The API is simple, having only a handful of methods but provides good, high performance security.

It provides everything needed for secure data transmission but unfortunately the library is relatively quite large, the Arduino Due has at it's disposal 512KB flash memory but the full NaCl library is 3MB. Fortunately the same creators along with Bernard van Gastel, Wesley Janssen and Sjaak Smetsers made TweetNaCl. Which is a tiny implementation of NaCl, still providing good performance and good security but with a significantly smaller code size of 40KB. It retains the same protections against timing attacks, cache-timing attacks, has no branches depending on secret data and no array indices depending on secret data. In addition it is thread-safe and has no dynamic memory allocation[8]. It is portable and easy to integrate, the library is easily added as it consists of two files, there is no complicated configuration to be set up or any dependencies on external libraries. Because of this compactness it is easy to read and understand it's operation. Although not as fast as NaCl it is still fast enough for most applications. The authors of the library feel that the performance is acceptable as"Most applications can tolerate the 4.2 million cycles that OpenSSL uses on an Ivy Bridge CPU for RSA-2048 decryption, for example, so they can certainly tolerate the 2.5 million cycles that TweetNaCl uses for higher-security decryption (Curve25519)." [9].

TweetNaCl is still small after compilation at 11KB thus reducing instruction cache misses. It is a full library and not a set of isolated functions, for a TweetNaCl application, only six functions are needed. crypto_box for public-key authenticated encryption; crypto_box_open for ecryption; crypto_box_keypair to create a encryption key pair; and similarly for signature keypairs crypto_sign_keypair, to sign messages crypto_sign and to verify signatures crypto_sign_open, and. It is open source and the developers encourage it to be used as much as possible.

Unfortunately the library won't compile as is on a Arduino, one problem is that there is no /dev/random and therefore no randombytes() which means that it can't create random secret keys and therefore good keypairs. The Arduinos can generate pseudorandom numbers by generating a seed from an analogue pin which will provide fairly random numbers but it is considered insecure for cryptographic applications to use pseudorandom

numbers like that[10]. It is possible to create true random numbers by utilising the truly random nature of atmospheric noise or background radiation but this requires extra components and is outside the scope of this project. Also, as mentioned in the implementation the Arduino can't use the C library as is, it needs to be converted into a C++ header file and cpp file relationship.

For public key authenticated encryption, TweetNaCl uses three components: Curve25519 Diffie–Hellman key-exchange function, Salsa20 stream cipher for message encryption and Poly1305 for message authentication. Curve25519 Diffie–Hellman key-exchange function is used to compute the shared secret between sender and receiver using the sender's secret key and receivers public key. Curve25519 is an elliptic curve, which is an approach to public key cryptography that is based on the algebraic structure of elliptic curves over finite fields, used in conjunction with the elliptic curve Diffie-Hellman anonymous key agreement protocol[11]. For message encryption the Salsa20 stream cipher encrypts a message using the shared secret. Salsa20 uses a pseudorandom function based on add-rotate-xor, bitwise additions and rotation operations. The cipher uses XOR operations, 32-bit addition, mod 2^{32} and constant-distance rotation operations on an internal state of sixteen 32-bit words[12]. The library uses Poly1305 as it's MAC function[13].

For key signature, TweetNaCl has a fourth component, Ed25519 public key signature system. Edwards curve digital signature algorithm is a variant of Schnorr signature based on Twisted Edward curves. It is faster than existing schemes but does not reduce strength of security[14]. TweetNaCl gets it's name because the developers tweeted all of the source code in a 100 tweets to prove how small it was.

2.3 Types of attacks

2.3.1 Replay Attack

When this attack occurs the attacker replays a valid message attempting to cause a legitimate action to be repeated. If Bob wants Alice to provide authentication and she duly provides some encrypted signature to prove her identity. Eve can capture that signature, she does not need to know what the signature is but she knows that it is a signature. She can then connect to Bob and when Bob asks for identification she can use this message to pretend she is Alice. To prevent this attack Alice might use an identifier that is only valid for one use, this could be session tokens or one-time passwords. These attacks are also known as playback attacks.

2.3.2 Man in the Middle Attack

In this attack there is an attacker between two parties, Bob and Alice, who wish to communicate. The man in the middle, Eve, changes messages as they are in transit to either pretend that she is the person that the other thinks they are talking to, change the contents of the message or something else. An example is if Alice asks for Bob's public key, Eve can capture that

public key, replace it with her own and send that and because Alice has no way to prove that it is Bob's key or not she accepts it. So when Alice sends a message that has been encrypted with what she thinks is Bob's key, Eve can take it, decrypt it with her key, change the message then encrypts it with Bob's real public key. When Bob receives the message he believes it is from Alice.

2.3.3 Bit-Flipping Attack

This is where the attacker can change the cipher text in some way that causes a predictable change in the plain text. Eve does not need to know exactly what the plain text is. For example if Alice was to send a message to Bob saying that she owes him £100 and Eve knows at least some of the message format, she can change the number at the end into £1000.

2.3.4 Stream Cipher Attack

If the same key is reused in a stream cipher then it becomes vulnerable. If Alice were to encrypt two messages of the same length with the same key, Eve can take the two encrypted messages and XOR them to produce A XOR B and thus find the message. If one message was longer than the other then Eve could only find out the section of the longer message that was the same length as the shorter.

2.4 Timing Attack

In this type of attack strategy, an attacker measures the time that the legitimate user takes to perform some action that involves the secret key. If the time taken depends on the key then it is possible to deduce information about the key. For example the login program in early versions of Unix executed the crypt function only when the login name was recognised. By this, an attacker was able to find out if a login name was correct even if the password wasn't.

2.5 Machine to Machine

M2M refers to the direct communication between two devices using any sort of channel. This is a component of IoT and when connected in this way small, low power sensors can transmit their data to another device which can collate, perform analysis or some extra computation or pass the data along again before it reaches a human user. Present day applications include monitoring the health of machinery, digital billboards or sensory networks. An application may possess multiple sensors in a network that pass sensor data to multiple nodes which do some computation, such as adjust machines to correct errors, replenish stock or manage a system and possibly sending information, such as state across the Internet to a human user.

2.6 Technologies used

In this project the Arduino was programmed using C++. The C++ was developed in the Arduino IDE. An Arduino Due, Arduino Uno, DS18S20 temperature sensor, resistor and two Ethernet Shield R2 boards were used. On the server side there is an SQL server, PHP scripts that accept that Arduino data and put it in the SQL server. And a Java web application that uses Java Database Connectivity, JBCD, to access the SQL server and outputs dynamic HTML when accessed. The Java code was developed using Eclipse Jee Mars. Bitbucket, using with Git, was used to remotely store all the software.

2.6.1 XAMPP Server

XAMPP, Apache, MySQL, PHP and Perl come together to form XAMPP which is free, open source, cross platform, web server stack package for the creation and testing of local server systems that can then be moved as they are onto live systems. This XAMPP used in this project also contains Tomcat.

2.6.2 PHP

PHP is a server-side scripting language used in web development. It is especially suited server-side applications as it is very portable. PHP code is contained in a file and when the server receives a request, the code is executed by the PHP runtime. It can be used with many OS's, web serves and relational database management systems. It is very common for web hosting providers to support PHP use by their clients.

2.6.3 SQL

Standard Query Language is a special purpose programming language for managing data store in relational databases. SQL organises data into databases and tables and has operations for inserting, updating, deleting, altering and querying data. The vast majority of the databases in businesses use SQL.

2.6.4 Apache Tomcat

Tomcat is an open source implementation of the Java servlet, JavaServer Pages, Java Expression Language and Java Websocket technologies. It provides the ability to produce dynamic HTML in a pure Java environment. It accepts web application archives, WAR, files for ease of updating web apps.

2.7 Secure Transmission of Keys

The problem is more acute with symmetric key encryption as it is the same key used to encrypt and decrypt a message, if the key is compromised messages can no longer be trusted. The most secure and high tech solution is to meet in person and in private with the person you plan to communicate

with and exchange keys there. If the communication is of a diplomatic nature, the bag used to transport the keys can have special legal protection against being opened and this is called a diplomatic bag[15]. You could send a key over another secure channel that you trust but there is always the chance that it has been compromised and it is not possible to verify that the person on the other side of the secure channel is who you think it is without first having a secure identifier which you can't get solely through internet communication. There is a similar problem with Asymmetric key encryption, a user can freely distribute their public key and they can be pretty sure that the messages they receive are secure but it is not possible to prove who has sent you the message without their public key signature but initial public key transmission are vulnerable to man in the middle attacks.

Chapter 3

Design

3.1 Aims

The basic aims are to design and implement a basic protocol for the transmission data to a server. This protocol will have security such that the messages are protected from modification, spoofing and can't be read in transit. There is to be a server application that allows the user to view the data being sent to the server. Following on, the needs of machine to machine communication will be considered and the protocol will be expanded to include direct communication and transmission of keys. Finally, as this will be on a low powered device, the power usage is to be measured to establish performance and power overhead of security.

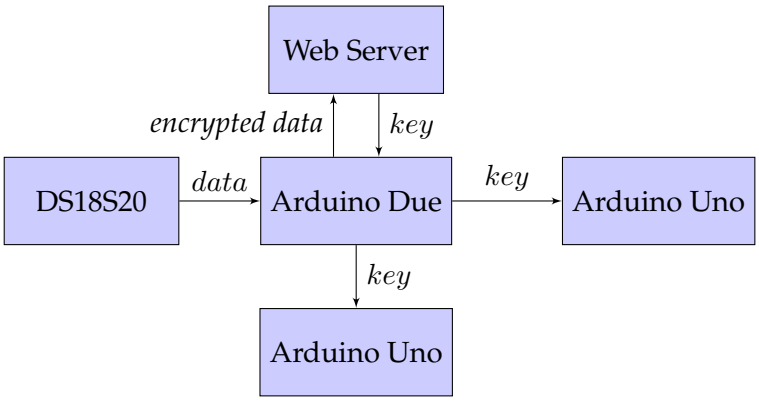

FIGURE 3.1: Node Diagram

3.2 Microcontroller

The first step in this application is the acquisition of data. For example data, a variable that is monitored for important applications in the real world was chosen, this being temperature. There are a range of temperature sensors out there but one that is compact, cheap, freely available, has a large temperature range, high precision and can derive power directly from the data line, so called parasite power, is the DS18S20. It returns a 9-bit byte array so is ready to encrypt straight away.

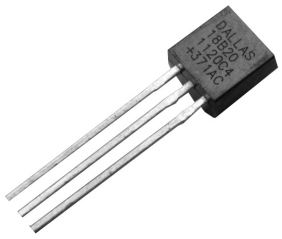

FIGURE 3.2: DS18S20 Temperature Sensor

The Arduino Due is a large Arduino, it is the first one with a 32-bit ARM core micrcontroller, has 54 digital I/O pins, 512KB flash memory, 96KB SRAM and a 84MHz clock speed. A lot of microcontrollers are 8-bit which means that they are limited in their cryptographic options. With a 32-bit architecture and a large amount of flash memory it is possible to have a light weight cryptographic library for our application but the Due is still low cost enough that it can be used for an IoT sensor application. Because it is an Arduino it benefits from the large community, wide range of compatible components and large set of open source libraries.

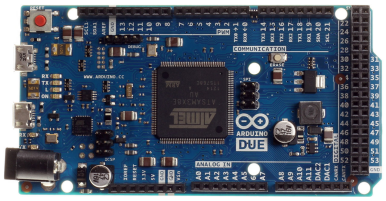

FIGURE 3.3: Arduino Due Microcontroller

Once the temperature data is on the microcontroller there needs to be a way to transmit the data. As the data is being sent over a network the natural choice is an Ethernet or WiFi shield. The Ethernet Shield was chosen as it is much cheaper than the WiFi Shield but completes the same job. The first revision of the shield is no longer made so the second revision, R2, will be utilised. It allows for easy connection of the Arduino to the Internet, it uses the Wiznet W5500 Ethernet chip, supports up to 8 different socket connections at a speed of 10/100Mb and has a MicroSD card slot to store network settings. It needs an different library than the first revision[16]. Fortunately the two libraries share the same API so software that was built for the first revision is compatible with the second. Simply swap put the older shield and library and replace with the newer one.

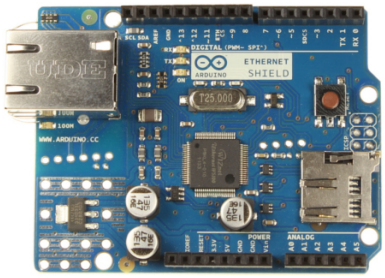

FIGURE 3.4: Arduino Ethernet Shield

3.3 Security

There are a lot of security options for desktop programmes and communications like AES, WEP and SSL. In this more ubiquitous field the implementations have been around for a while but for microcontrollers only in recent times have they become powerful enough at a cheap enough price and therefore popular enough for developers to write or adapt cryptographic libraries suited for microcontroller communications. The library in this application had to have acceptable security, really as close as possible to encryption systems in more powerful computers but still have acceptable performance, have a small enough code size to be stored on the microcontroller and be easy to use. Of course, it needs to prevent attackers from reading or altering the data and proving who sent it. Some microcontroller applications have extra chips that are solely for encryption but that comes at extra cost.

TweetNaCl was chosen for this project. It is a public-domain, auditable and tiny implementation of NaCl, a high-speed cryptographic library. NaCl aims for absolute performance with good security at the expense of portability whereas the aim of TweetNaCl is portability, small code size and auditability. The developers claim it is first auditable high-security cryptographic library. It is a recently created library, released in 2014 and it provides public-key cryptography, secret-key cryptography, hashing and string comparison. TweetNaCl is a full cryptography library but has only two files, needs little to no memory to be stored, has similar performance to NaCl and is easy to use. It has a set of high level methods that require the necessary variables and returns the encrypted or signed message. It doesn't provide many options, you call the method with the message and keys and the software does the rest. This reduction in customisability reduces the likely hood of a mistake by a developer using the library.

3.4 Server

3.4.1 Web Server

The web server is needed to be able to complete the relatively difficult jobs of decryption, checking signature and integrity, pulling data from the SQL database and displaying the data. For this reason and the fact that there is a Java implementation of TweetNaCl from a developer called Ian Preston,

a Java Server Pages application was set up running on Apache Tomcat. JSP is a tool to dynamically create web pages using Java code. JSP could be joined with the Java implementation of the C library, TweetNaCl, to retrieve the encrypted values from a database, decrypt and display them. For the database structured query language, SQL, was chosen. When the data is being sent to the web server it needs to be sent in a particular way. Data is passed into and taken from servers is using POST and GET requests. And it is possible to send a GET or POST request to a file and that file then executes. When the Arduino is sending data to the database it was decided the script it sent a POST request to would be a PHP file as taking the encrypted and signed data and storing it could be handled by a simple PHP script.

3.4.2 Arduino Server

If it is desired that the temperature be taken from more than one location then there will need to be multiple nodes on the network. Or if there needs to be extra microcontrollers on the network for some other task. One such task is the distribution of public keys, if the server is to update it's public key, say as part of a regular update schedule to provide good security or if a key is known to be comprised then it will need a way to pass on the key. The Due can request a new key or the server can indicate to the Due that a new key has been made available and the Due will make a request for that key then make a connection to the other Arduinos in the network and send them that key. An extra Arduino Uno was chosen as an extra node for testing. The network will be built using Ethernet and be an extension of the Arduino Due to web server relationship. As the Arduino Due is a client to the web server the other nodes will need to be hosts if information is to be passed between them. The setup described in figure 3.1 is desired.

Chapter 4

Implementation

4.1 Overview

The Arduino Due asks for the server's new public key and for a new nonce that it should use for the following temperature data transmission. It does this by sending a GET request to a JSP page on the server. The public key and nonce are newly generated by the server and the nonce is encrypted using the keys and nonce used in the previous transmission. The Due is connected to the DS18S20 temperature sensor and it receives the temperature data over the OneWire protocol. This data is signed using the Arduino's secret signature key and then encrypted using the Arduino's secret key, the Server's new public key and the new nonce. The encrypted data is sent as a POST request to the add.php file on the apache server which executes it and the first thing add.php does is call connect.php which has the SQL database details and makes a connection to the database. Following that add.php builds up a SQL query that inserts the values sent in the post request into the appropriate table entries and then the connection is closed. When a user want to view the files, they use a browser to send a GET request to the Java web app. The app builds up a SQL query to take out all the values from the database, decrypts them and verifies the signature before printing out in a table. When it comes to public key transmission the Due sends a POST request to the Arduino Uno that is set up as a web server. The Uno recognises that it is receiving a POST request and stores the key.

4.1.1 Temperature reading

The DS18S20 temperature sensor is wired up with a 4.7Ω pull up resistor, between the orange wires, on the bread board shown in figure 4.1. A pull up resistor is a resistor between the sensor and the positive power supply so that the signal will be a valid logic level if external devices are disconnected or a high impedance is introduced. It is connected to digital pin 9, yellow wire, because it can't be on pins 10, 11, 12, 13 as they will be used by the Ethernet Shield. When looking at the temperature sensor the furtherest left pin is V_{dd} and normally this would be connected to the Arduino's 3.5v or 5v output but the DS18S20 is in parasite power mode which scavenges power off the middle data line, DQ. When the DQ pin is high some of the charge is stored in a capacitor that will be used to power the device when the data is being read. In parasite power mode both the GND and V_{dd} are connected together, by the bluewire, and then to ground.

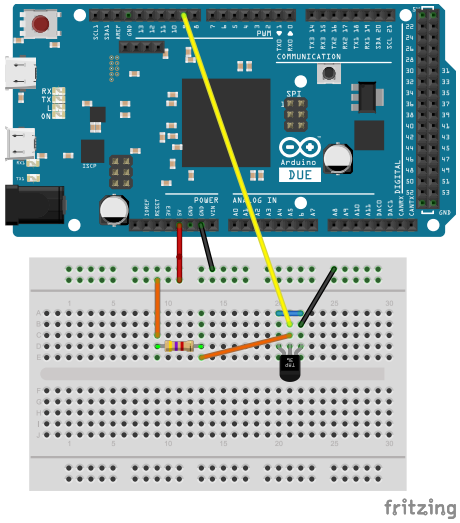

FIGURE 4.1: DS18S20 in parasitic power mode connected to Arduino

The DS18S20's scratchpad memory is divided up into 9 bytes as shown in figure 4.2. The first two bytes are the ones of most interest as they store the actual temperature value. Bytes 2 and 3 are the high and low trigger registers, these can be used to trigger some action when the temperature goes above or below a certain threshold. Bytes 4 and 5 are only for internal use and cannot be overwritten. Bytes 6 and 7 can be used to calculate the extended resolution. Byte 8 holds the cyclic redundancy check which is an error detecting technique. The bytes with asterisks are values which stored in non volatile EEPROM, the rest are stored on volatile SRAM[17].

SCRATCHPAD (POWER-UP STATE)		
Byte 0	Temperature LSB (50h)	(85°C)
Byte 1	Temperature MSB (05h)	
Byte 2	T_H Register or User Byte 1*	
Byte 3	T_L Register or User Byte 2*	
Byte 4	Configuration Register*	
Byte 5	Reserved (FFh)	
Byte 6	Reserved	
Byte 7	Reserved (10h)	
Byte 8	CRC*	

FIGURE 4.2: DS18S20's memory organisation

The library used to read the DS18S20 is OneWire, which is a proprietary library from Maxim that performs half-duplex bidirectional communications between a host/master controller and one or more slaves. As seen in the following code snippet, figure 4.3. The command *ds.write(0x44,*

1) starts the internal A-D conversion operation. Once this process is finished the data is copied to the Scratchpad registers. A delay is needed to charge the capacitor and to ensure conversion is complete before reading the data. Which is done with *ds.write(0xBE)* and then the data is read out using *ds.read()* and put into an array.

```
1                OneWire ds(9);
2                  ...
3                ds.write(0x44, 1);
4                delay(1000);
5
6                present = ds.reset();
7                ds.select(addr);
8                ds.write(0xBE)
9
10               Serial.print(" Data = ");
11               Serial.print(present, HEX);
12               Serial.print(" ");
13               for ( i = 0; i < 9; i++) {
14                    data[i] = ds.read();
15                    Serial.print(data[i], HEX);
16                    Serial.print(" ");
17               }
```

FIGURE 4.3: DS18S20 temperature sensor Arduino Code

4.2 Cryptographic

Before the data can be signed and encrypted the new nonce and server public key have to be requested. The Due sends a GET request in a very similar manner to the POST request shown in figure 4.6 and the server returns, byte by byte, the information. The Arduino takes the char arrays and converts them into two byte array. The public key is sent unencrypted so a human user must check that it is the same key in this demo application but the nonce is sent encrypted and signed. Shown in figure 4.4 if this is the first transmission then the nonce will have been encrypted using the presintalled keys and nonce else the server public key nonce and generated by the server previously are used. The way the Arduino knows if this is the first transmission is if the byte array *serverpkold* is empty as after the temperature data is encrypted the old nonce and server public key and stored in another array so they can be used to decrypt the next nonce. Throughout this application the same signatures are used. The Due decrypts the nonce using the relevant keys and removes the 32 bytes of leading zeros that needed to be placed by the server for successful encryption before verifying the signature in line 12. Nonces are 24 bytes in length and a signature is 64 bytes in length.

```
#include <TweetNaCl2.h>
TweetNaCl2 tnacl;
int Suc_Decrypt;
int Suc_SignVerify;
int scnlenwz = crypto_box_NONCEBYTES + crypto_sign_BYTES +
    crypto_box_ZEROBYTES;
byte scnoncenewtemp[scnlenwz]; //signed encrypted nonce
    with zeros
byte snoncewz[scnlenwz]; // signed nonce with zeros
int snlenwoz = crypto_box_NONCEBYTES + crypto_sign_BYTES;
byte snoncewoz[snlenwoz]; //signed nonce without zeros
int snlen = crypto_box_NONCEBYTES;
byte nonce[snlen];
  ...
if(serverpkold[0]==NULL){
  Suc_Decrypt = tuit.crypto_box_open(snoncewz,
      scnoncenewtemp, scnlenwz, preinstallnonce,
      preinstallserverpk, arduinosk);
}else{
  Suc_Decrypt = tuit.crypto_box_open(snoncewz,
      scnoncenewtemp, scnlenwz, nonceold, serverpkold,
      arduinosk);
}

Suc_SignVerify =
    tuit.crypto_sign_open(nonce,&nlen,snoncewoz,snlenwoz,serverpksign);
```

FIGURE 4.4: Decrypting the next nonce

Now that the server public key and nonce have been found the secure temperature data transmission can take place. Much like the server, when the Arduino is encrypting a signature and message it must have padded out the data with 32 bytes of leading zeros specified in the NaCl website. Take special care that exactly 32 bytes are placed in front because if there isn't the correct number the encryption process will still complete without errors. However the decryption will fail and it isn't apparent that that error occurred when data was encrypted. The code used to sign and encrypt the data is shown in figure 4.5.

```
1  #include <TweetNaCl2.h>
2  TweetNaCl2 tnacl;
3
4  byte arduinosk[crypto_box_SECRETKEYBYTES] = {...};
5  byte arduinosksign[crypto_sign_SECRETKEYBYTES] = {...};
6  byte serverpk[crypto_box_PUBLICKEYBYTES] = {...};
7  byte nonce[crypto_box_NONCEBYTES] = {...};
8
9  int const messageLength = crypto_sign_BYTES + 9;
10 byte message[messageLength] = {...};
11 unsigned long long signedMessageLength=0;
12 byte signedCipher[signedMessageLength];
13 unsigned char
       signedMessage[messageLength+crypto_sign_BYTES];
14
15 tnacl.crypto_sign(signedMessage, &signedMessageLength,
       message, messageLength, arduinosksign);
16 tnacl.crypto_box(signedCipher, signedMessage,
       signedMessageLength, nonce, serverpk, arduinosk);
```

FIGURE 4.5: TweetNaCl Arduino Signature and Encryption Code

The C TweetNaCl library has been converted into an Arduino library. The Arduino lanugage is based on C and C++. It therefore needs an instance of TweetNaCl created which in the sketch is called *tnacl*. With this instance the methods needed can be accessed. In this prototype some keys are preinstalled, the Arduino's secret key, it's secret signature key, the server's public signature key and the first nonce and first server public key that will only be used once to decrypt the first nonce. With each transmission the Due will get a new nonce and server public key.

Lines 4-13 set up the message byte arrays that are to be passed between the methods. *message* is initialised as a byte array of size *messageLength* which can has to be *crypto_box_ZEROBYTES*, 32 bytes, plus the length of the message. *signedMessage* and *signedMessageLength* are passed in by reference so after *crypto_sign()* is complete the message with the signature and the size of that array will be in those variables, respectively. The resulting signed message needs to have 32 bytes of leading zeros added to it before encryption. The function *crypto_box* completes that takes and the places the cipher into *signedCipher* which is passed in by reference.

4.3 Data transmission

Once the data has been encrypted it is to be packaged up and sent across the network.

```
#include<Ethernet2.h> //Ethernet Shield R2 library
#include<SPI.h>

byte mac[] = {
0x90, 0xA2, 0xDA, 0x10, 0x2D, 0xD6 //of the Ethernet Shield
};
char server[] = "192.168.0.6"; //IP of apache web server
EthernetClient client;

IPAddress clientIP(192, 168, 0, 30);

 if(Ethernet.begin(mac)==0){
     Serial.println("Failed to assign IP");
     Ethernet.begin(mac, clientIP);
  }else{
    Serial.println("Assigned IP");
  }

if(client.connect(server,80)){
    String data = "temperatureHex=";
    int contentLength = data.length()+final.length();
    Serial.println("Connected");
    client.println("POST /tempLog/add.php HTTP/1.1");
    client.println("HOST: 192.168.0.6");
    client.println("Content-Type:
        application/x-www-form-urlencoded");
    client.print("Content-Length: ");
    client.println(contentLength);
    client.println();
    client.print("temperatureHex=");
    client.print(final);
   }else{
      Serial.println("Failed to Connect");
   }

   client.stop();
```

FIGURE 4.6: Ethernet interfacing and transmission Code

Just before the cipher is sent the byte array is converted into a String so that it can be passed around easily as one parameter. This is completed simply by cycling through the byte array and adding each entry together. Care has to be taken when there are hexadecimals that are 0x0F or lower. The leading zero is lost during the conversion to String which means that when the cipher reconverted into a byte array, it is incorrect and cannot be decrypted. This is solved by explicitly adding an extra "'0" String if the byte is less than or equal to 0x0F. The device needs an IP address which is completed through Dynamic Host Configuration, DHCP by the router. The

router running DHCP dynamically allocates network configuration parameters such as IP addresses to devices so that they automatically get one that isn't in use. This is completed with the line *Ethernet.begin(mac)* which returns a 1 on successful IP allocation or 0 on failure. If it fails then a static IP can be assigned manually by *Ethernet.Begin(mac, clientIP)*. A connection to the server is attempted using the IP and the port number, 80 in this case. If it successful the information is transmitted as a POST request, a POST request is a request method in the HTTP protocol. When a server receives a POST request it knows to take the data and complete an action with it. The request makes it known that it wants add.php to be executed upon receiving the data. Once the data is sent the connection can be closed and other actions performed on the Arduino.

4.4 Server Side

For the prototype, an Apache and Tomcat server with SQL was set up using XAMPP on a desktop. The Arduino causes the add.php to be run and the first thing that it does is make a connection to the SQL server using SQL IP address, username, password and the name of the database. If it can't connect it abandons the task and returns an error message. On successful connect it returns the connection variable.

```
1  <?php
2      include("connect.php");
3
4      $link=Connection();
5
6    $temp=$_POST["temperatureHex"];
7
8    $query = "INSERT INTO tempLog (tempHex)
9      VALUE ('".$temp."')";
10
11     mysql_query($query,$link);
12     mysql_close($link);
13
14     header("Location: index.php");
15 ?>
```

FIGURE 4.7: Arduino to SQL interfacing code

The data to be stored is extracted out of the POST request and placed in a variable. Then an SQL query is built up before being sent to the SQL server and the connection terminated. The SQL table contains an ID, the time at which the temperature data was received and the data and is created using the command shown in figure 4.8. Notice between the database creation code and the PHP file the corresponding variables, tempLog, the table name and tempHex, the data.

```
CREATE TABLE `iotplatform`.`tempLog` ( `id` INT(255) NOT
    NULL AUTO_INCREMENT , `timeStamp` TIMESTAMP on update
    CURRENT_TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ,
    `tempHex` VARCHAR(100) NOT NULL , PRIMARY KEY (`id`))
    ENGINE = InnoDB;
```

FIGURE 4.8: SQL database creation code

The data stored in the database is still encrypted, now a way is needed to decrypt it and display it to the user. A Java web app, using Java server pages, JSP, was created as there are Java implementations of the TweetNaCl library, among other variations, available[18]. Eclipse JEE Mars was used to create a dynamic web project which extends HttpServlet for the creation of dynamic web pages. In this you override at least one of the following methods, doGet, doPost, doPut and doDelete so depending on the type of request received a different action will occur. This application has the code in the doGet as the server will receive a get request when it is accessed by a user.

4.4.1 Decryption and display of temperature data

```
1 protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
2 response.setContentType("text/html");
3 PrintWriter printWriter = response.getWriter();
4 printWriter.println("<html>");
5 }
```

FIGURE 4.9: Prepping the client printer in JSP

Using the code shown in figure 4.9 the headers and tags you would need and expect in a HTML page can now be printed dynamically much like printing to a console. The keys are defined similarly to the code on the Arduino except the server has it's own secret key and the Arduino's public key.

```
final String DB_URL="jdbc:mysql://localhost/iotplatform";
String user = "root";
String password = "";
try
      {
        // Register JDBC driver
        Class.forName("com.mysql.jdbc.Driver").newInstance();

          // Open a connection
          Connection conn =
              DriverManager.getConnection(DB_URL, user,
              password);

          // Execute SQL query
          Statement stmt = conn.createStatement();
          String sql;
          sql = "SELECT id, timeStamp, tempHex FROM
              tempLog";
          ResultSet rs = stmt.executeQuery(sql);
          // Extract data from result set
          while(rs.next()){
            //Retrieve by column name
            int id = rs.getInt("id");
            String tempHex = rs.getString("tempHex");
            Timestamp timeStamp =
                rs.getTimestamp("timeStamp");
  }
}
```

FIGURE 4.10: Accessing the SQL database in JSP

In figure 4.10 is a section of code running in tomcat. The java application was created with Eclipse Mars and was exported as a WAR file before being store in the tomcat directory. The app is given the SQL details before entering a try catch that prints whatever the error message is to the client. Java database connectivity, JDBC, is an API for client access to a database. Before the java app is exported as a WAR file it must have the JDBC bin jar in the lib folder in WEB-INF for the eclipse project otherwise this won't work. The app gets a new instance of JDBC and then opens a connection to the server before building up a query to pull out the contents of the table. The variables are put into a result set which can be used to access each individual variable with the string name as a parameter. The encrypted message is still in it's String format and needs to be converted back into a byte array. During the conversion to a String on the Arduino half of the leading zeros are lost but they are added again before the conversion to byte array. The API in the Java implementation of is compatible with the TweetNaCl C API but provides another level of abstraction on top.

```
1 TweetNaCl.crypto_box_open(signedMessage, cipher,
    messageLength, nonce, arduinoPublicKey,
    serverSecretKey);
2 byte[] javaPlainTextMessage =
    TweetNaCl.crypto_sign_open(signedCipherArray,
    ArduinoPublicSignatureKey);
```

FIGURE 4.11: Decryption of message and verification of signature in JSP

The security needs to be taken open in the reverse order as to how it was put on. The nonces and secrets keys used have been stored in the order they were created. Which corresponds with the order of the data in the SQL database. As each entry from the table is removed the corresponding secret key and nonce , from the 2D arrays, is used. *TweetNaCl.crypto_box_open* passes the decrypted message out by reference but for *TweetNaCl.crypto_sign_open* the message without the signature is returned. This is an example of the top layer of abstraction in Ian Preston's implementation of TweetNaCl. The *TweetNaCl.crypto_box_open* method is the exact same as the the C library but *TweetNaCl.crypto_sign_open* is a slightly built upon method that has fewer parameters, completes more work behind the scenes and returns the byte message. The variable *javaPlainTextMessage* is the temperature in plain hexadecimal but it still needs conversion to integer so it can be read by the average user. The code is taken from the DS18S20 example from Arduino.

The web app upon being accessed decrypts and checks the signature of each entry, using the keys that it has stored, in the table before converting the raw hex temperature data into more readable integers and displaying in a simple HTML table that can be accessed by the user. When the Arduino has data to send it will make a POST request to a PHP file on the Apache server which takes the data given to it and places it in the SQL server.

4.5 Public Key Transmission

For a secure connection public keys must be sent. At the start of the Arduino due's code it sent a GET request, very similar to the POST request shown in figure 4.6. The request is sent to a Java page called Secret Key. This page is similar in set up to the page a user accesses to view the temperature data. When accessed the page calls (TweetNaCl.crypto_box_keypair()) to create the server's secret key and corresponding public key and the public key. It stores the secret key in a 2D array and prints out the public key to the Due with two terminating characters either side. Along with the next nonce, this process is explain in section **??** Once this request has been sent the server sends all of the information, byte by byte. Which also includes information about the server, data and time. During debugging this can be used to print out the state of the web application as it is being run. The code used to extract the the public key is very similar to the code shown in figure 4.12 expect it watches for the character < to signify that the next word will be the key. Afterwards the key needs to be converted back into a byte

array from string before it can be used. The key is then used to encrypt the next temperature data transmission before being copied over into a second array so that it can be used to decrypt the nonce that will be sent encrypted from the server for the next secure temperature data transmission.

For the creation of other nodes on the network that are connected onto the Due, an Arduino Uno was set up as a host and example node. As two clients can't directly connect together with the Ethernet protocol and since the Arduino Due is a client to a web server, the Uno must be a host. The Uno will be a server similar to the XAMPP web server expect it isn't directly connected to the internet. The Due sends a POST request to the Uno, very similar to the one it sends to the XAMPP web server, in figure 4.6 however the key is under the content header. The method that the Due uses, sending a POST request that executes a php file to extract the data from the POST cannot be used, without difficulty??, as PHP can't be run on an Arduino web server. Instead there is an extra header in the POST, *"Content: key"*.

```
String incomingWord = " ", key = "";
int saveNextWord = 0, takeData = 0;
 if (client.available()) {
       char c = client.read();
       Serial.write(c);
       if(c == ' '){
         key = incomingWord;
         incomingWord=" ";
       }else{
         incomingWord = incomingWord + c;
       }
       if((incomingWord=="Content") && (takeData)){
          saveNextWord = 1;
       }
       if(incomingWord=="POST"){
         takeData=1;
       }
}
```

FIGURE 4.12: Handling POST requests in Arduino

When setting up the Uno server, explicitly define what gateway and subnet the router being used has as the defaults *Ethernet.begin(mac)* uses aren't always correct. The server sits open on a certain port, 8081 in this case, and waits for clients to make a connection. Once they have, the server sends an acknowledgement that it has received a connection and starts reading in the request. There it watches out for the word POST so that it knows to store the content and, of course, for the content so it knows what to store. The request from the client is read out a byte at a time so it is necessary for those bytes to be made into strings so specific keywords can be looked out for. If the byte is not a space then it is part of a word so it is added to the subsequent bytes until another space is reached then it is considered a word and compared against. Once the key is taken it is in String format and needs to be turn back into hex and store as the key to be used in further encryptions.

4.6 TweetNaCl Library

The TweetNaCl library as it stands in it's original form is not compatible with Arduinos. The C library compiles without errors but the compiler warns that the TweetNaCl method names are undefined and as a result do not perform their tasks. The Arduino language is a mix of C++ and C so it doesn't support every aspect of each language. When the methods are accessed they simply returns random numbers, it is possible that it is trying to access some area of memory, not finding the correct function and simply returning whatever it finds. It is not understand why this is the case but to get the libraries to work it is a simple case of converting the library into C++ syntax. With a header file that has the main methods used in the project and the #defines and a cpp file with the TweetNaCl code. This is added in the same way to the Arduino IDE and in the code an instance of the class is created and methods are accessed with the dot operator. In this application not every function in the TweetNaCl library will be used so any methods that weren't going to be utilised were not copied over into the converted library.

Chapter 5

Strength Of Security

5.1 Sign then Encrypt

Generally encryption stops unauthorised parties from accessing information, assuming authorised parties are the only party with access to a secret key, prove message integrity. Signatures prove that the message came from a known sender and the sender cannot deny sending the message. These are two distinct operations and the order of these operations matters. If a user was to encrypt a message then sign and send it, as the signature is added to the message, it can be stripped off by an attacker and replaced. If the attacker managed to intercept the transmission of public keys in the first place then they can now impersonated the real recipient. Signing after encryption defeats the purpose of signing, which is to prove that the owner of the signature saw the data and cannot disprove that if their signature is attached to it. If an encrypted message is signed then it is entirely possible that the owner of the signature was not aware what was in the message. In this application the message is signed first so the hash is taken of the plaintext message and it can be proved that the signer was aware as to the contents of the message and then it is encrypted[19].

5.2 Storing data in plaintext

Storing vital data such as passwords and usernames in plain text in a database is generally considered quite a bad idea. This provides one point of failure that if broken means that every password stored is now useless and the accounts are wide open. In some circumstances that initial break might not seem so severe, say if that is for a forum site where the worst action that could be taken was writing an inappropriate message but users sometimes use the same password for many different sites. Although in this application the information being stored in temperature data and not passwords but the security implications are still there. This application would have that single point of failure that is so deplorable. It is necessary to provide as many layers of security as possible rather than have one hard layer. That being the security of the database. As such this application stores all the temperature data as it arrives, in it's encrypted form. This way the keys will need to be known by the attacker in order to gain the data.

5.3 Public Key Transmission

If the keys are preinstalled they can be used to sign and encrypt new public keys before they are sent. So that the receiver can use the presinstalled keys to prove integrity and authenticity of the new keys. If there are no preinstalled keys then there is no way to prove that the public key being received is indeed the public key sent. At least for a computer. The most secure and safe method to make sure the keys are the same is to have a human user manually inspected the key before sending and after sending and carefully ascertain that the one sent is correct. If there has been a man in the middle, MITM, and either the key has been altered or replaced entirely then the human user can see the difference.

5.4 Bit Flipping

Bit flipping is the process of changing parts of the encrypted message so that it says something else. This is especially potent when the format of the message is know. As the format of the message is know to us, a test can be set that if it succeeds would change a number. In this project a test was set up to demonstrate the cryptography caused detected that the message had been altered in transit. After a signed message was encrypted it was altered with random hexadecimal numbers and it each time the decryption process would fail. The encryption process adds a hash of the message and the decryption process would find a different hash and would fail each time.

5.5 Timing Attacks

TweetNaCl is an example of constant time software, which means that the time of execution does not depend on secret data and is therefore not vulnerable to timing attacks. To have this quality TweetNaCl avoids all loads from address and all branch conditions that depend on secret data and it is thus inherently protected against cache timing attacks.

5.6 Replay Attacks

This attack is the act of resending of captured valid messages to repeat a valid action, like sending money or authentication. The potential for serious damage by a replay attack here is limited as this application. If an attacker resends some data to the server, say a malicious set of temperature data, when the server would go to decrypt that message it would try to use the incorrect nonce and public key and the message would be ignore, thus protecting the application from replay attacks.

5.7 Length of cipher text

Is it possible to gain information about the message from looking at characteristics of the cipher text? In bad cipher text it can be possible to find

out the types of message that are sent as the resulting ciphertext may have change in a certain, predictable. For example if the message is a large number in contrast to a low number, there may be noticeable differences. Unfortunately a vulnerability exists in the cryptography chosen but it is a common weakness and one that does not have much consequence. That is the length of the cipher text corresponds with the length of the message. This is offset somewhat by the addtition of the signature so the cipher text is much larger than one would expect if it was know that 9 bytes of hexadecimal were sent. To fully get around this the message is padded out with zeros so that the whole message is always forty one bytes in length. Which does mean the message has a limit of forty one bytes unless the source code is altered. To test how the cipher text changes when sending different messages four encryptions took place. The messages were YES, NO, 1 and 9999.

Plaintext Message	Encrypted Message
YES	0xAE, 0x43, 0xCD, 0xA5, 0x8E, 0x54, 0xE9, 0x30, 0x59, 0xB1, 0xD5, 0xA5, 0xBF, 0x24, 0x9D, 0xEE, 0x69, 0xDB, 0x37
NO	0x2E, 0xBC, 0xCC, 0x6B, 0x9E, 0xDB, 0x29, 0x64, 0xF6, 0x26, 0x49, 0xEF, 0xE9, 0xFA, 0xBD, 0x9C, 0x7E, 0xD1

TABLE 5.1: Resulting ciphers for encrypting words

From theses message it is possible to tell the number of bytes sent if you know that the encryption method adds 16 extra bytes but other than that this test shows there isn't a discernible difference between encrypting YES and NO.

Plaintext Message	Encrypted Message
00	0x5B, 0x74, 0x76, 0x4C, 0xF4, 0x19, 0x65, 0x37, 0xC4, 0x53, 0xAF, 0xB0, 0xCE, 0x18, 0x1, 0x1, 0x30, 0x9E
1	0x49, 0x1B, 0x1E, 0x52, 0x10, 0x38, 0xD7, 0x38, 0x30, 0x65, 0x71, 0xBC, 0xEE, 0x65, 0x3D, 0x6, 0x31, 0x9E
99999	0xC, 0xC, 0x29, 0xE2, 0x4E, 0x81, 0x67, 0x5, 0x78, 0x49, 0x8C, 0xA1, 0x4F, 0x69, 0x8, 0xBB, 0x9, 0xA7, 0x5D, 0x63, 0x4D

TABLE 5.2: Resulting ciphers for encrypting numbers

Again, the bytes return from encrypting the test case of numbers don't indicate that the integer contained in the message is increasing but it does show the length of the number.

Chapter 6

Results

The basic objectives were to sign and encrypt a message in order to stop an attacker from reading the messages, to protect against them being altered in transit, to authenticate them, to send that message to a web server and have that data be accessible by a user.

6.1 Basic Objectives

It can be seen that the DS18S20 temperature successfully found the temperature of the room. That that data was given a signature and then encrypted. It was then packaged up and sent as a POST request across the network to a web server that stored the data in a database. Then the user could access that database and view the temperature data. The message could not be read or altered as it was encrypted nor would any message that wasn't signed by an authorised user be displayed in the web app.

6.2 Power Consumption

As this is to be a low power system that might run on batteries for a good amount of time, it is important to analyse the power consumption of the device. Although saving power wasn't a top priority concern in the creation of the application, there is more that could be done to save power such as completely shut down modules on the Arduino Due that are never used. The temperature won't be taken every second so between bursts of activity the Due could be put into a sleep mode and be woken up on an interrupt rather than the busy wait implemented in this prototype.

To start the power was recorded as the temperature was being taken with the DS18S20 temperature sensor, in both parasite power mode and regular power mode. It was found that during both operations the power consumption was the exact same at 0.123A and 0.613W. This result was not surprising as the power consumption of the temperature sensor is so small compared to the Due that even though it doesn't consume power in parasite mode, the overall consumption of power is barely affected.

To isolate the the power consumed when the Due was only signing and encrypting, the Ethernet shield was left unplugged. This way the effect of the encryption would be seen without the effects of powering an Ethernet shield throughout. The readings upon start up were 0.123A and 0.613W, when the data was being given a signature the readings were 0.17A and 0.970W and when the message was encrypted it rose to 0.20A and 1W.

In addition the power consumption when the Ethernet shield was attached but had no cable plugged was recorded. The power consumed during the signature and encryption process was the same as above but during DCHP the readings were .199A and 0.990W as it tried to gain an IP address from the router. Once the loop iteration was complete, the code went straight into a busy loop which was a tiny drop in power at 0.198A and 980mW. A busy wait is a waste of power and resources so at this point the Arduino Due should go into a sleep mode and be woken up by an interrupt when it was to record the temperature again.

and fucking windows fucking firewall

The next test was with the shield plugged into the router with the Ethernet cable. The power consumption shot up to 0.265a and 1.30W during DHCP at the start of the sketch and there it remained even during the encryption process. Evidently the shield requires such a significant of the power that by comparison the cryptographic library is as good as unnoticeable. There are versions of the Ethernet Shield with an extra module that allows power over Ethernet. This extra module doesn't add a significant increase to the overall cost to the shield but can alleviate battery problems and extend the life of batteries by extracting power from the Ethernet cable that is connected to the router

The final test was to find out if there was a difference in power consumption depending on the length of message sent. Messages of size 20 and size 250 were sent with no appreciable difference in power consumption found.

The voltage through the experiments was the exact same, a steady 4.96V which makes sense as this is round about what USB 2.0 can give out and is well within normal limits for the Arduino Due.

6.3 Additional Objectives

6.3.1 Transmission of public keys

Expansion of the network was considered with multiple devices communicating directly. This objective was reached by adding an Arduino Uno with another Ethernet shield as a webserver and the Due could pass on data, such as new server public keys, to it. If more nodes where desired then more Uno webservers could be added and the Due would simply go through the list of servers and send a POST request to each to update their keys. At present the Due passes on the server's public key that it receives when it receives it. jkhasdassadjkhsadjkhdsajkhasd

6.3.2 Secure transmission of nonces

Instead of using nonces that can be predicted, say counters, or worse reusing the same nonce this application encrypts the nonce to be used in the next temperature data transmission using the nonce from the previous nonce or in the initial case a set of preinstalled keys, that are never used again, are utilised. The nonces are created randomly on the server and then sent securely, this ensuring that the nonces for the secure temperature data transmission can never be guessed and protecting against replay attacks.

Chapter 7

Critical Evaluation

With the way it is set up at the moment the XAMPP server pages, add.php and connect.php, and the JSP application that displays the temperature data are accessible through a browser by anyone on the network. Realms could be set up or IP filtering. A realm is a database of usernames and passwords that identify valid users of a web app and can be used to provided levels of access.

The public and private key of the server and the public key of the arduino are stored as plaintext in the web application and it might be possible to read them from the WAR files. This is not as much a problem for the public keys, if caught quickly, as it means we can no longer trust that new messages are from an authorised source but if the secret key is leak then all messages sent from the Arduino using that key pair are compromised.

The messages are padded with zeros to avoid leaking out the message length. Perhaps it would be better to include random data rather than a set of never changing zeros. Thus the rare case that an attacker notices that the first 32 bytes don't change or worse that they are all the same would be avoided.

The Arduino doesn't have access to /dev/random and can't provide good enough random data to make good key pairs. Which means that it can't update it's own secret key, for example if the key has been compromised or it is part of a regular renewal service to key security tight.

When the public keys are transmitted for the first time, a human user is required to manually inspect the keys, in private, and ensure they are the same before they can be used for data transmission. This is, unfortunately, unavoidable as there is no way to prove that this key that arrived is the one that was sent. The internet is an unsecure platform and it cannot be trusted. There are many things that could happen as it transits the web.

The PHP files that the Arduino Due sends a POST request to do not authenticate the requests and as such anyone who knew the ip address of the server could send data into the SQL database. However when the Java web app retrieves the data out of the database if it is not encrypted with the keys then the decryption will fail and the data won't be displayed to the user.

The JSP page that sends a new public key and encrypted nonce to the Arduino Due does not have any restrictions on it's access. Therefore is a malicious connection is made to the server then that connection will receive the data. This public key and nonce could then be used by the attacker to add in their own data or to simply interfere with the normal operation of

the application. This could be solved in a similar manner to securing the PHP files with IP filtering or setting up a realm.

If a encrypted temperature message doesnt make it, then the sync is lost

This prototype has used the Ethernet Shield as it was much cheaper than a WiFi shield. If an implementation similar to the one described in this report was to be implemented then it would be convenient to provide connection over WiFi. The product also risks not being adopted as more and more products have wireless capabilities and users see it as the norm and resent cables. This will be especially important if the device uses batteries to power itself.

Chapter 8

Conclusion

This report has shown that is it possible to provide low power, not overly complicated encryption and authentication on a micrcontroller based IoT system. TweetNaCl is an excellent library that can be added to any project easily, has tiny code size but still provides acceptably strong encryption at a quick speed. It is a full library and can produce asymmetric and symmetric encryptions plus a SHA-512 hashing function and a string comparison function which any system can implement to cover it's encryption needs.

Any computer application which handles information that is private or could be used to do damage such as a persons credit card details meaning that person loses money or the lose of private user data that causes the loss of trust and damage to the reputation of a company needs good encryption. As more and more devices are connected to the internet such as smartphones, smart televisions, smart domestic appliances and anything else an IoT engineer thinks might benefit from an internet connection. As the entire planet shifts more and more into the digital world, this need grows ever larger. When data, such as credit card details, are lost this may be a huge problem but it is not fatal. However with the advent of heavy machinery such as cars becoming part of the internet of things, and other examples no doubt soon to follow, security should be at the forefront of everyone's minds. This project has aimed to show that acceptable encryption can be implemented easily, even on low power systems. In this prototype, the users private temperature data is safe from attackers.

The Arduino was a good choice as Arduinos have been use for fast prototyping and for teaching people who might not have a background in electronics or computing how to create projects, such as IoT projects. There a multitude of internet connected projects for the Arduino, many connect to benign objects such as the LEDs on the board but they can be connected to many more things, some which can be troublesome if compromised. Security is generally an afterthought, for experienced and novice developers alike. Even massive companies with millions pounds worth of revenue make security blunders or just don't bothered. It is hoped that this project will add to the list of secure, internet connected Arduino projects available so that one of the first things of everyone's mind when starting a project is security. The powerful but lost cost Arduino devices prove that it is possible to have good quality security in any IoT system and more security options will pop up as the price to performance ratio continues on it's upward trajectory. Which is very beneficial to the IoT industry as a whole.

Security measures will never be fully secure. Given infinite computer resources and infinite time any security algorithm can be hacked and there

might bugs, backdoors, employees willing to leak secret keys. However if the algorithms make it unfeasible to attack or simply not worth the attackers effort required then that is as good as fully secure. At the moment there are so many devices that simply have no protection that there is an open source project google powered search engine that shows all the devices that are vulnerable[20]. These devices lack even rudimentary security features and even a casual attacker with not too much knowledge or experience could gain access and control. By having even the minimal amount of protection, obviously the more protection the better, you can dissuade would be attackers because they have plenty of easier targets. Or make it so that the reward of breaking into your system simply is not worth the effort.

The TweetNaCl library satisfies all the requirements of a good microcontroller cryptolibrary. It is open source and the developers encourage it's use where ever possible. It was used with success in this project as the users private data could not be read in transit nor altered in transit and the receiver could prove that a certain person sent the message.

Bibliography

[1] Matt Warman. *50 billion devices online by 2020*. 2012. URL: http://www.telegraph.co.uk/technology/internet/9051590/50-billion-devices-online-by-2020.html.

[2] Nicole Kobie. *Hacking the Internet of Things: from smart cars to toilets*. 2014. URL: http://www.alphr.com/features/389920/hacking-the-internet-of-things-from-smart-cars-to-toilets.

[3] Andy Greenburg. *Hackers Remotely Kill a Jeep on the Highway—With Me in It*. 2015. URL: http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[4] Jill Scharr. *Hackers Hijack Prius with Mac Laptop*. 2013. URL: http://www.tomsguide.com/us/hackers-hijack-prius-with-laptop,review-1797.html.

[5] Jeremy Korzeniewski. *Tesla Model S successfully hacked by Zhejiang University team*. 2014. URL: http://www.autoblog.com/2014/07/18/tesla-model-s-successfully-hacked-by-zhejiang-university-team/.

[6] Which. *Which? probe uncovers Hive heating app data risk*. 2015. URL: http://www.which.co.uk/news/2015/08/which-probe-uncovers-hive-heating-app-data-risk-407275/.

[7] Frank Denis. *Introducing Sodium, a new cryptographic library*. 2013. URL: https://labs.opendns.com/2013/03/06/announcing-sodium-a-new-cryptographic-library/.

[8] Daniel J. Bernstein et al. "TweetNaCl: A crypto library in 100 tweets". In: (2014), p. 3.

[9] Daniel J. Bernstein et al. "TweetNaCl: A crypto library in 100 tweets". In: (2014).

[10] Arduino. URL: https://www.arduino.cc/en/Reference/Random.

[11] Daniel J. Bernstein. *A state-of-the-art Diffie-Hellman function*. URL: https://cr.yp.to/ecdh.html.

[12] Daniel J. Bernstein. *Snuffle 2005: the Salsa20 encryption function*. URL: https://cr.yp.to/snuffle.html.

[13] Daniel J. Bernstein. *A state-of-the-art message-authentication code*. URL: http://cr.yp.to/mac.html.

[14] Daniel J. Bernstein et al. *A state-of-the-art message-authentication code*. URL: http://ed25519.cr.yp.to/index.html.

[15] Boleslaw A. Boczek. *Internation Law: A Dictionary*. Scarecrow Press INC, 2005.

[16] Arduino. URL: http://labs.arduino.org/Ethernet+2+Library.

[17] Maxim Integrated. *High-Precision 1-Wire Digital Thermometer*. 2015. URL: datasheets.maximintegrated.com/en/ds/DS18S20.pdf.

[18] Ian Preston. *tweetnacl-java*. 2016. URL: https://github.com/ianopolous/tweetnacl-java.

[19] Don Davis. *Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML*. 2001. URL: http://world.std.com/\~dtd/sign_encrypt/sign_encrypt7.html.

[20] Zakir Durumeric et al. *A Search Engine Backed by Internet-Wide Scanning*. 2015. URL: https://censys.io.