POSTQUANTUM CRYPTOGRAPHY IN MEDICAL SMART DEVICES

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Electrical Engineering

By

Melvin Relf

2022

SIGNATURE PAGE

THESIS:	POSTQUANTUM CRYPTOGRAPHY IN MEDICAL SMART DEVICES
AUTHOR:	Melvin Relf
DATE SUBMITTED:	Fall 2022
	Department of Electrical & Computer Engineering

Dr. Mohamed El-Hadedy Aly
Thesis Committee Chair
Electrical & Computer Engineering _____

Dr. Anas Salah Eddin
Associate Professor
Electrical & Computer Engineering _____

Dr. Halima El Naga
Professor
Electrical & Computer Engineering _____

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Mohamed El-Hadedy (Aly) for his time and effort on helping me throughout my thesis always giving energy to approach more challenges. I would like to thank the love of my life Sachi for always supporting and putting up with me during this entire process. Special acknowledgements also go to Dr. Ayesha Khalid from Queens University Belfast, United Kingdom. Also special thanks to Russell, Lino, Juan and Michael from the Cal Poly Pomona undergraduate/graduate lab Their contributions made this project possible. Finally, I would like to thank my father and mother for their incredible support throughout my entire career.

ABSTRACT

As using the Internet has become more commonplace, more and more devices seem to rely on it to connect to other devices and receive updates. Although smart devices have helped many operate more efficiently than before, they are also very autonomous and are susceptible to outside breaches. Whether the healthcare sector is ready or not, IoT and smart devices will continue to become more integrated with the workforce. This is why security must be an upmost priority when designing medical devices.

By utilizing the Internet, patients may be monitored and managed continuously and simultaneously. Currently, some of the medical IoT devices from which health data can be obtained include Pulse SpO2 sensors, BP sensors, Glucometers, EMG sensors, EEG sensors, and GSR sensors. Unfortunately, many of these devices use outdated encryptions schemes that can be defeated by simple brute force attacks. It is not only important for medical devices to be protected from current threats but also future threats. This paper will explore the future of security in medical IoT devices and how post quantum cryptography can be utilized.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: BACKGROUND

The Internet of things (IoT) is defined as physical hardware that connect and exchange data with other physical devices and systems over the Internet or other communication protocols. IoT enables people and objects in the physical world, to be able to interact with each other in ways that were not previously possible. IoT devices have been implemented in home maintenance, security systems, transportation, energy grids and smart healthcare systems [11].

IoT is on the rise and could revolutionize the healthcare sector in terms of social and economic benefits. By implementing synchronized cloud based computing and communication, healthcare frameworks (including individual patients, medical devices, and medicine dosages) may be monitored and managed continuously and simultaneously. Currently, some of the medical IoT devices from which health data can be obtained include Pulse SpO2 sensors, BP sensors, weight scales, ECG sensors, Glucometers, EMG sensors, EEG sensors, and GSR sensors[12].

The IoT's connectivity provides a new way to store, monitor, and use healthcare related data on a daily basis. The increasing cost of hospital care and the increasing prevalence of chronic diseases as lifespans increase immediately demand the evolution of healthcare from a hospital-centered system to a person-centered environment. It has been predicted by some that in the next few decades, healthcare will shift from being hospital-centered, to hospital-home-hybrid in the 2020s, and then ultimately to home-centered in the 2030s [13]. Even if healthcare remains to be hospital-centered based, it is essential

that the application and integration of IoT systems to be constantly monitored and improved[11].

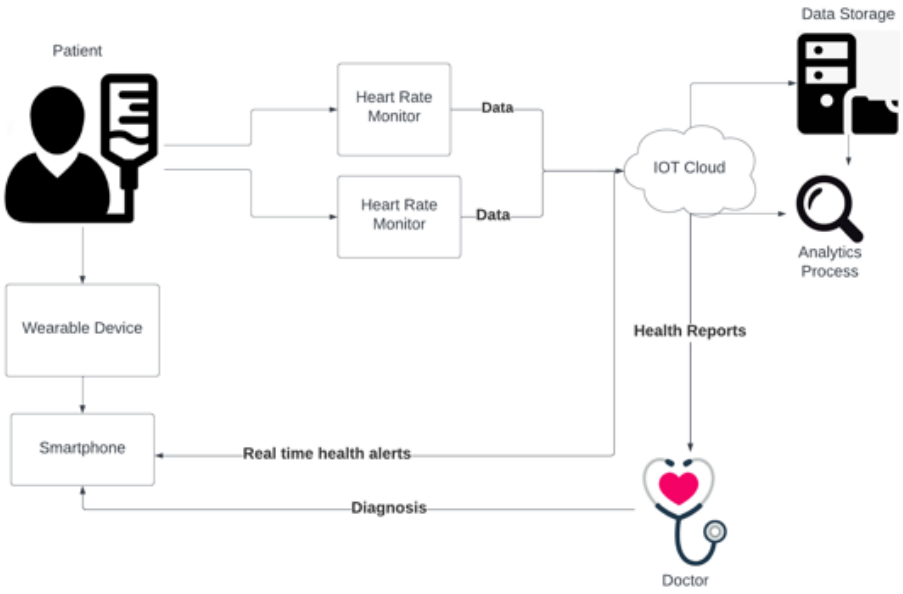

Figure 1: Example of medical IoT environment.

1.1 Internet of Things in Healthcare

Healthcare IoT systems are distinct from other computer systems in that they are built to serve human beings, which inherently raises the requirements of safety, security, and reliability. Improving a patient's quality of life is important to mitigate the negative effects of being hospitalized. Healthcare IoT systems must also be able to provide real-time notifications and responses regarding the status of every patient. The development of personal devices such as smartphones and tablets has helped establish a basis for mobile healthcare. Modern smartphones are often embedded with an assortment of sensors that can track and record a patients' location, activity levels, heart rate, blood oxygen and even blood pressure [14]. However, these devices are not entirely reliable and only can provide basic information for quick reference. Countless wearable devices

have recently been developed to extend the medical capabilities of mobile devices. Wearable devices have a variety of functions, including data acquisition from sensors, preprocessing of biometric data, file storage, and data transfer to internet-connected clients. These devices also offer significant advantages to medical doctors by allowing them to remotely interact with patients. These interactions can include diagnosing a disease, giving treatments, and even adjust medicine dosages. Although IoT systems sound like a perfect match for hospital systems, they also raise concerning questions and introduce new challenges involving the vulnerability of the system and the privacy of doctors and patients [11].

One of the main and most obvious problems in IoT systems is the increasing number of devices that are being connected to the Internet. The more devices that are connected to the same node, the more the performance of the system degrades due to limited computer resources such as bandwidth and processing power. However, this degraded infrastructure is unacceptable in a real life healthcare environment. Also, a large portion of these devices themselves have limited resources due to being lightweight mobile devices. Therefore in order to meet necessary requirements, the mobile components in the healthcare IoT system require a reliable communication architecture [15].

Figure 2: Examples of wearable IoT medical devices, Source: EAI Endorsed Transactions on Pervasive Health and Technology, 2018

1.2 Wearable IoT

Wearable IoT (WIoT) is defined as a wearable device or sensor that enables the monitoring of personal data, including vitals, mental, and behavioral data to enhance a person's daily quality of life. Due to the direct involvement of human beings in WIoT, providing secure, accurate and rapid communication among medical devices, shared hubs, and the medical providers is vital. Although WIoT devices appear to be harmless devices intended to benefit its user, privacy must be an essential part of the architecture [18]. For example, some medical devices collect personal information, such as the user's location and movement activities. If this information is not safeguarded during the process of storage or communication, the patient's privacy or even safety may be compromised. There can also be a chance of societal fear caused by the concern that the

government or private companies are using such devices for tracking people or selling sensitive data. [13].

Internet Protocol (IP) enabled devices in a Medical Sensor Network (MSN) can the private data of patients to remote healthcare providers. In such cases, the transmitted medical data may be routed through an untrusted network infrastructure, such as a public Wi-Fi service. Reliable and efficient techniques are needed to protect against attacks from these kind of networks [11].

Keeping track of high risk patients has always proved to be a challenging and expensive task for health care services. Chronic disease management make up about 1/3 of the United States healthcare budget with most of that funding is related to heart disease, asthma and diabetes. Remote monitoring with IoT enabled devices will enable physicians to constantly monitor high risk patients. IoT in healthcare is also known as IoMT (internet of Medical Things). IoMT is basically medical equipment with internet connectivity that can communicate with another device. This kind of device should also be capable of linking with servers or cloud based services to store measured data so it may be analyzed later [17].

1.3 Real World Examples

Remote patient monitoring (RPM) is an emerging area where IoT is being used in a healthcare context. RPM (also known as Telehealth) is a type of healthcare service that allows patients to use a mobile device to perform a routine checkups and then send the results back to their healthcare provider instantaneously. This technology incorporates medical devices like a glucose meter, blood pressure monitor or a heart monitor [19].

This data can be delivered to a physician's office via a software application. This technology is meant to reduce hospital admissions or doctor visits. This kind of technology saves patients from traveling to a remote location whenever they have a simple medical question or update in condition. It also reduces the cost of keeping a patient checked in to hospital room which can be very costly. The Kaiser Family Foundation found that in 2013 the average daily cost in care for a single patient was over $1,700 [17].

The Body Guardian Remote Monitoring System is another system that provides the functionality of remote medical care. The system also addresses several security concerns one would have. First, it isolates the patient identification information (PIN) and observed data [18]. Then the device encrypts the measured data both during transmission and while in storage. This technology is used quite often with the two groups that often require the most substantial amount of medical attention: those who are chronically ill and the elderly. The medical providers can keep a close eye on their patient's vitals and symptoms and even intervene if necessary [17].

FCC's National Broadband Plan mentions that remote patient monitoring technology can save the healthcare industry $700 billion over the next 15 to 20 years. *Mobile Health (mHealth)* Devices that can be worn by patients can also send information to their providers [21]. Infusion pumps that connect to analytic software in hospitals have numerous sensors that measures the condition of the patient. With Internet connectivity, even medication can be administered to a patient from anywhere including at home [20].

1.4 Integration of Medical Devices

A pacemaker is a medical device which is implanted under a person's skin, with electrical wiring that goes down to their heart. They are used to help regulate abnormal heart rhythms. They have improved immensely over time and recent advancements have allowed pacemakers to have more digital capabilities. This includes transmitting data from the patient's heart to nearby access points or servers. In the current age of cybersecurity and hacking, it is easy to see how detrimental it would be to have such sensitive information compromised. Access point devices, which collect information about the patient's health while at home, sends the data to remote servers in the hospital. This can greatly benefit a patient who has mobility issues. However, the communications protocols used in older or out of date devices can be very trivial and is susceptible of being attacked by hackers [17].

Implantable cardioverter defibrillator (ICD) is a battery-powered device placed under the skin that keeps track of a patients' heart rate. More than 10,000 patients receive these defibrillators annually in order to prevent a sudden heart attack. An *insulin pump* is another device that has benefitted from IoT. These small devices can be programmed to release small doses of insulin continuously, or a single dose close to mealtime to prevent a dangerous sudden rise in blood glucose. A functioning insulin pump should imitate the body's natural process of releasing insulin [22].

Concern about the vulnerability of medical devices like a pacemakers, ICDs, insulin pumps, defibrillators, ultrasound monitors and scanners is growing as healthcare facilities increasingly rely on devices that connect with the Internet or with other devices.

As of 2018, the Department of Homeland Security is investigating potential vulnerabilities in about two dozen medical devices. Hospital medical devices may be vulnerable to attackers especially if the devices haven't been updated with the latest security protocols [17].

1.5 Security Concerns

In 2015, two cryptography researchers discovered over 68,000 medical systems that were openly revealed on the Internet . The major concern with this research was that these almost all of the devices involved were connected to the Internet through computers running very old versions of Windows XP [23]. These devices were discovered by using the search engine Shodan, which can find IoT devices that have connected to the internet. These accounts are often susceptible to brute-force attacks and are using hard-coded logins. The two researchers were alarmed to discover anesthesia equipment, cardiology devices, nuclear medical systems, infusion systems, pacemakers, MRI scanners, and other devices using simple search queries [17].

The two researchers also created and uploaded honeypots, which are special servers that appears can appear as any IoT device similar to an emulated device. These devices had false medical data and vulnerabilities similar to those of other medical devices. When the researchers reviewed and analyzed the honeypot logs, they discovered that hackers managed to authenticate via secure shell protocol (SSH) the fake devices over 55,000 times with almost 300 malicious payloads. If the hackers could figure out that the devices could direct them to other servers with more important data, they would not hesitate to perform a more thorough attack. This experiment also demonstrated that

hackers could use the devices to spread dangerous malware and viruses throughout a hospital's IT systems [23].

There has been a rise in data security and liability risks in the healthcare sector because of IoT. The Internet of Things brings many of the same security and privacy issues, but it is a much greater risk because these devices act autonomously. It is theoretically possible for hackers to remotely tamper with medical devices to harm individuals, but a case of this occurring has yet to be discovered. Devices are more commonly hacked so attackers can get into larger medical systems and steal protected health or bank account information [19].

In June 2015, a report was released by TrapX, which coined the term "medjacking". The report provided details about multiple incidents of medjacking in multiple hospitals. In the first case, a blood gas analyzer infected with two types of malware was used to steal passwords from the hospital system. At another hospital, the radiology department's image storage system was used to gain access to the main database. In the third and final case, hackers used the vulnerability in a drug pump to gain access to the hospital network infrastructure. A little known fact is that stolen medical identities are much more valuable to hackers than a credit card number. The current state of security in many medical devices allow hackers to easily access a number of healthcare provider's systems [25].

The Animas OneTouch Ping, was launched back in 2008. It was a wireless device patients can use to order the pump to deliver a dose of insulin. It was typically worn under clothes and was difficult to reach for some [24]. The company behind the product

Johnson & Johnson eventually informed patients that they have learned of a security vulnerability in one of its insulin pumps that someone could use to overdose a patient with insulin [19]. This system is vulnerable because communications were not encrypted, so a hacker could deliver unauthorized insulin injections. Given the nature of the device, the possibility of being hacked in this manner was low but it was the first time a major manufacturer of medical devices had openly issued a warning to patients about a cyber vulnerability in a medical device. This announcement also raised increased concerns about possible bugs or vulnerabilities in other medical IoT devices [17].

In another case, the Food and Drug Administration (FDA) in 2017 issued warnings about bugs in infusion pumps from a company called Hospira. This was the first-time FDA has issued a warning for a medical device based on a cyber security risk. In Aug 2015, the FDA recommended that all hospitals across the country should stop using a medical device that is vulnerable to cyber-attacks over a connected network [17].

1.6 Securing IOT in Healthcare environments

There are several basic security actions that providers and manufacturers of IoT devices can take including *encryption* and conducting a *secure boot*. A secure boot is making sure that when a device is turned on, none of its configurations have been modified [26]. The main challenge faced by the existing security solutions is how to provide end to end encryption in a way that would allow parties to securely and efficiently communicate with each other beyond the local network boundaries.

Cryptographic keys can be generated within the devices or networks on the fly via the usage of information collected by medical sensors when and as needed. The generated

keys can be employed in end-to-end communications to securely encrypt/decrypt data transmitted between medical sensors and health caregivers. The keys can also be used for authentication and authorization of peers in healthcare IoT systems. Enabling mobility and extended freedom for healthcare IoT systems offers a high quality of medical service, as it allows patients to have a greater sense of independence [11].

CHAPTER 2: ENCRYPTION AND POST QUANTUM CRYPTOGRAPHY

Before discussing cryptography, it is important to discuss what quantum computing is and the future risks it may bring. Quantum computers are incredibly powerful machines capable of performing calculations magnitudes faster than classical computers. A conventional computer stores data using binary data but in quantum computing stores data using qubits. These computers can orient the qubits using the properties of quantum physics, resulting in extremely fast processing power. Although current quantum computers are too small to outperform current computers in practical applications, they will pose an even more significant threat to encryption in the near future[3].

In 2016, the National Institute of Standards and Technology (NIST) started taking submissions for new cryptography schemes that could potentially replace current encryption standards [9]. As a result, several new encryption methods were proposed. Ever since then, the interest in quantum resistant algorithms has only increased. Post quantum algorithms fall into these major categories: hash, code, lattice, multivariate, and super-singular [4].

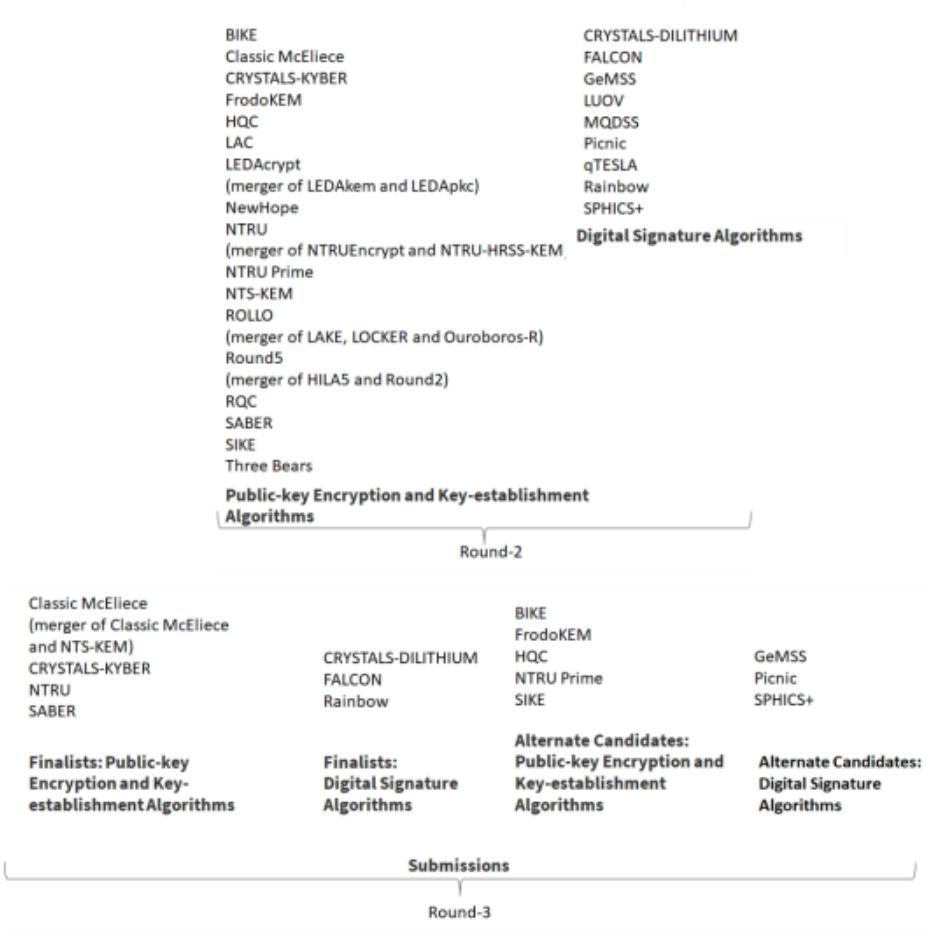

Figure 3: Round 2 and 3 Submissions for NIST, Source: Securing the Future Internet of Things with Post-Quantum Cryptography, 2021

Current encryption standards such as AES and RSA have long been thought to be very robust and secure. But in 1994, an engineer named Peter Shor developed an algorithm that allowed RSA to be breached only a few years later. Currently RSA-3072 is the recommended standard although, 90% of all online encrypted connections still use RSA-2048. But increasing the key size has its limits and it is predicted that RSA reaches its limits at a key size of 4,096. The discovery of the Grover algorithm proved that quantum computers have a quadratic increase in speed when searching databases compared to traditional computers [5].

As quantum computer becomes closer to becoming a reality, researchers are continuously trying to improve or replace the security in IoT applications. Although these challenges mostly theoretical in the present time, there is still no guarantee that current post-quantum cryptography-based IoT applications will be capable of resisting attacks from quantum computers [2].

2.1 Hardware Limitations of IoT Devices

Resource-restricted IoT networks and devices use much smaller key sizes ranging from 128 bits to 4096 bits. This is a problem as post-quantum algorithms require much larger key sizes. Therefore, integration of IoT networks with post-quantum cryptography algorithms require optimization for key sizes, network performance, and scalability. IoT medical devices can introduce latency during cryptography procedures at both the server and client sides. For example, there is a limit on the number of signatures that a cryptosystem uses. So if a device must generate new keys for subsequent messages, the device will consume a large amount of energy and will not give the most efficient results [27].

The majority of post-quantum cryptography systems focus on proving robust security but give less attention to other parameters including energy consumption, communication latency, and resource consumption which are equally important for IoT devices. Presently, there is no universal standard to measure the effective security levels of current post-quantum algorithms against future quantum attacks. Optimization of quantum algorithms for IoT nodes is also necessary for IoT devices [27]. In lattice based cryptosystems there is a need to speed up the polynomial-based multiplication calculation

since it has a time complexity of $O(n^2)$. The use of nodes and parallel processing could reduce the energy consumption and execution time of such expensive calculations. Additionally, there are a set of mathematical tasks in other post-quantum cryptography schemes (such as finite fields) that need optimization for IoT devices [28].

IoT based medical companies are mostly focused on developing lightweight, smaller and smarter products without as much consideration for the security [29]. In the near future, quantum threats to standard cryptography in smart IoT healthcare systems will also be plausible. Thus, post-quantum preventative measures are required to be considered while designing the security architecture for IoT medical devices. The strong security, wide applicability, and efficiency to protect against attacks make the lattice-based cryptosystem a prime candidate for the lightweight medical devices [27].

A pseudo-random key generator is often utilized in a variety of cryptosystems. This random key generation process usually requires average-case intractability or hard problems [30]. The worst-case reduction to the average-case hard problem requires a selection of proper parameters that are easier in lattice-based cryptography for smart IoT devices since a smaller size is required for smart IoT. Lattice-based algorithms also use matrices or vectors in small order fields or rings with small size parameters. The uniqueness of this property of lattice-based algorithms make them more appropriate for security proposals. The challenge of proposing a security solution for IoT networks and medical devices using lattice-based cryptosystem is the goal of combining everything that requires lightweight cryptography protocols suitable for both resource heavy and resource

restricted smart IoT devices. Examples of lattice-based cryptography include NTRUEncrypt, R-LWEenc, NewHope and CRYSTALS-Kyber.[27]

2.2 KYBER-Crystals

Lattice cryptography initially gained a lot of interest in the theoretical community because designs for cryptographic constructions were accompanied by proofs based on worst-case instances of lattice problems. The first lattice-based encryption scheme was introduced by Ajtai and Dwork [31]. Eventually, the algorithm was further improved and shifted towards the Learning With Errors (LWE) Problem. This was relatively simple to use in cryptographic cases and asymptotically at least as hard as some standard worst-case lattice problems [32]. The LWE assumption states that it is hard to distinguish from uniform the distribution (A, As + e), where A is a uniformly-random matrix in $Z^{m \times n}_q$, s is a uniformly-random vector in Z^{n}_q , and e is a vector with random "small" coefficients chosen from some distribution. Applebaum et al. showed that the secret s in the LWE problem does not need to be chosen uniformly at random: the problem remains hard if s is chosen from the same narrow distribution as the errors e [33].

The number of bits that can be transmitted is related to the dimension of the ring, thus using a ring Rq of larger degree n allows one to transmit more bits, and this is the main reason that Ring-LWE encryption is more efficient than LWE encryption. On the other hand, having a smaller k implies more algebraic structure, making the scheme potentially susceptible to more avenues of attack. Nevertheless, at this point in time, it is unknown how to exploit the algebraic structure of Ring-LWE and concrete parameters are chosen according to the corresponding LWE problem of dimension $k \cdot n$ [34].

2.3 TinyJAMBU

JAMBU is a lightweight authenticated encryption mode submitted to the CAESAR competition [35]. JAMBU is the smallest block cipher authenticated encryption mode in the CAESAR competition, and it was selected to the Third Round of the competition. TinyJAMBU mode is based on a keyed permutation. The state size of TinyJAMBU is only two thirds of that of JAMBU, the message block size of TinyJAMBU is half of that of JAMBU mode. The authentication security of TinyJAMBU mode is better than the Duplex mode when a nonce is reused [36].

TinyJAMBU a lightweight variant of JAMBU which is a 128-bit keyed permutation with no key schedule. The permutation is based on a 128-bit nonlinear feedback shift register. This lightweight permutation is used in the TinyJAMBU mode. The keyed permutation supports three possible key sizes: 128 bits, 192 bits, 256 bits. For the applications in which a fixed key is embedded in the devices, and for the applications in which a secret key is stored in a device for protecting multiple messages, TinyJAMBU uses only a 128-bit register for authenticated encryption [37].

CHAPTER 3: DIFFIE-HELLMAN KEY EXCHANGE

Wireless communication and messaging services have become very popular and essential to day to day life. In the days of analogue communication, the only concern was if the person on the other side of the line was actually who they said they were. But unfortunately, modern messaging protocols require end to end encryption or else the message can be intercepted while it moves from one party to the other. The Extended Triple Diffie-Hellman (X3DH) protocol has been widely used as the basis for key exchanges and server communication. In this section I will overview how the X3DH protocol functions and how it was implemented in this experiment [43].

But in order to understand how X3DH functions, one must have a basic understanding of the Diffie-Hellman protocol. Diffie-Hellman is a key exchange protocol dating back to 1976. It was conceived by Whitfield Diffie and Martin Hellman and is a means for two parties to calculate a shared secret from an exchanged key [38]. In order to simplify the algorithm, I will refer to the two systems that are trying to exchange messages as Alice and Bob. I will refer to a malicious third system as Mallory who is trying to intercept the message between Alice and Bob [43].

Diffie-Hellman utilizes a generator g and a multiplicative group or elliptic curve group of order q. The first step involves Alice and Bob choose private keys a and b respectively. These numbers are used to calculate the public keys A and B. The equations for each are displayed below:

$$A \equiv g^a \bmod q.$$
$$B \equiv g^b \bmod q.$$

Figure 4: A and B in Diffie-Hellman, Source: Radboud University, 2022

Alice then send *A* to Bob while Bob send *B* to Alice. Now the shared secret *K* can be independently calculated on both sides. This is due to the fact that $K \equiv A^b \equiv g^{ab} \equiv g^{ba} \equiv Ba \bmod q$ [43].

The security of the Diffie-Hellman scheme is very dependent of the choice of the group and generator. If they are too small solving the computation becomes relatively easy and if they are too large it becomes computationally expensive. In order for Mallory to compromise the system, they can either intercept the private key of one party or calculate the shared secret *K* manually using only the public keys [39].

There exists multiple versions of the Diffie-Hellman protocol. The public and private keys can either be static or ephemeral. Static keys, as the name suggests, remain unaltered by either party for all instances of communication. In this case the keys are precalculated outside of the protocol. As you can imagine, static keys are easier to implement but if a private key is compromised, then the entire system is compromised. On the other hand, ephemeral versions of Diffie-Hellman generate a unique pair of keys during each step of communication. These keys are much more secure compared to static ones especially during lengthy communication sessions. For ephemeral keys, the key generation must be part of the protocol [40].

The Diffie-Hellman protocol could also be breached if Mallory pretends to be Alice by sending Bob their own public key. This is why it is important to implement a means to authenticate public keys. This can become difficult to implement if the keys are replaced often or the communication channel is between a large amount of parties. Signature and certificate schemes are also effective means of ensuring that the exchanged message or key is from a trust worthy party [41].

3.1 The X3DH Protocol

The Extended Triple Diffie-Hellman or X3DH is a key agreement protocol that facilitates communication between multiple parties through an unprotected channel. X3DH was designed with asynchronous communication in mind which was a limitation of the Diffie-Hellman protocol [42]. This means it is possible for Bob to publish a message to a server, go offline and have Alice receive the message or shared secret using the information present in the server. In order for X3DH to be effective, a trusted server with a secure connection need to be established. In this project, Alice and Bob generate their own keys without the need of a third party server. The basic functionality of how the X3DH protocol functions has not been altered [43].

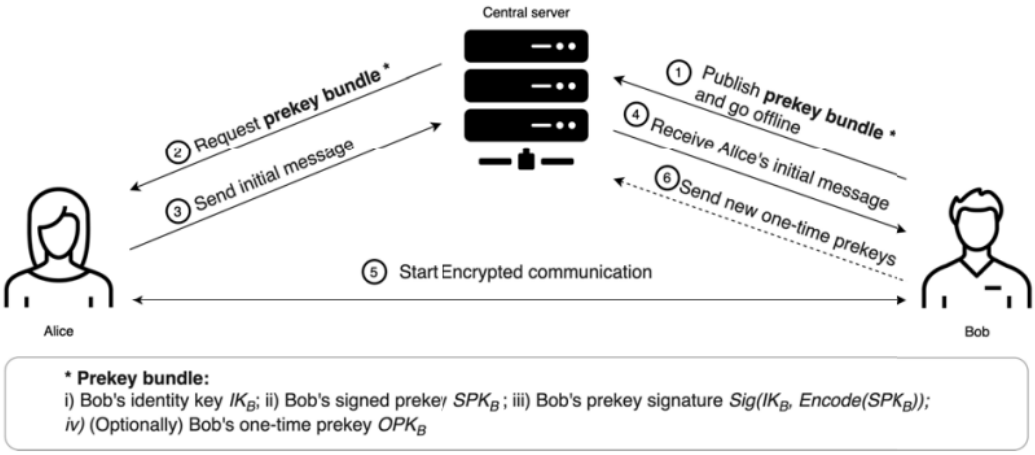

Figure 5: X3DH Protocol, Source: Radboud University, 2022

The first step in the Extended Triple Diffie-Hellman protocol is the generation of a prekey bundle. Bob creates a private key ik_b and public identity key IK_B and publishes that to a server. The public key is tied to Bob's identity and therefore is constant through the duration of the communication session. The next key that Bob must generate is known as the signed prekey or SPK_B. This prekey gets its name from the fact that Bob must store a certificate of SPK_B that is verified using IK_B. The signed prekey is replaced for every message sent but the previous signed prekey must be stored temporarily in order to decrypt the previous message. Depending on the implementation, signed prekeys may or may not be reused [43].

If the prekey is not reused it is known as a one-time prekey or OPK_B. One time prekeys are deleted by the server after a single use. If Bob sees that a one-time prekey has been reused, he will disregard the incoming message. Since these keys are uploaded to the server prior to the initiation of the X3DH protocol, it is important to ensure there are enough OPK_B's to last for the duration of the communication cycle. This concludes the

keys that must be generated on Bob's side. One time prekeys were not used in my experimentation [40].

In order for Alice to begin the message exchange with Bob, she must first request a prekey bundle. Alice needs to verify the certificate with Bob before she will be able to utilize the prekey bundle. Since she has access to the public key IK_B , Alice can check the contents of the certificate. If the contents are equal to the signed prekey SPK_B , then the certificate is authenticated. If the certificate is invalid, then Alice will reject the message all together.

After authenticating the certificate, Alice then creates an ephemeral Diffie-Hellman key pair with public key EK_A. Ephemeral keys are unique for each new instance of the security protocol and are only used once for every instance. Identity keys and public keys can be used for multiple instances of the protocol, but ephemeral keys are only used for a single instance. Similar to Bob, Alice also has her own identity key IK_A of which she has the private key. Alice has access to the following private keys: ik_a and ek_a . Now she can compute the shared secret keys that are needed to calculate the shared secret key K_{AB} [43]. Using a key derivation function KDF, Alice and Bob can subsequently calculate the shared secret key: $K_{AB.}$ In this context, || symbolizes the concatenation of the Diffie-Hellman keys. After Alice has calculated K_{AB} she deletes her ephemeral key, ek_A, and all the *Ki* [41].

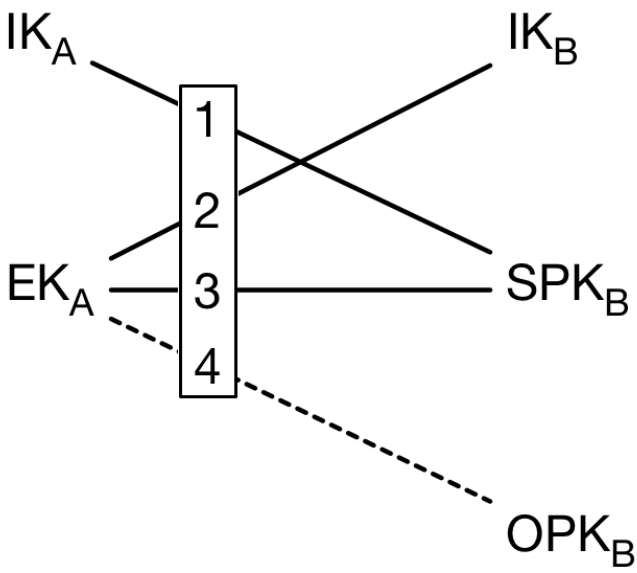

Figure 5: X3DH Protocol illustrated, Source: Radboud University, 2022

Now that Alice has calculated the shared secret, she can send an initial message to Bob so that he may calculate K_{AB} also. Alice will next send her identity key *IKA*, ephemeral key $EK_{A,}$ a unique identifier if a one-time prekey is used and an initial ciphertext that was encrypted with the shared secret key K_{AB}. The encryption method used is dependent on implementation and is up to the user to decide which to implement. After the first message is sent, K_{AB} may be used to derive future keys [40].

In order for Bob to decipher the ciphertext, he must also calculate K_{AB} . After receiving the first message from Alice, Bob would know her ephemeral public key and public identity key. He also has access to his own private keys: spk_B and ik_B. Bob can now do the same Diffie-Hellman calculations that Alice did, because the public keys required have already been sent by Alice. Finally, Bob can decrypt the first message received by using the same encryption scheme as Alice. It is important to implement an algorithm that checks for the validity of the decrypted message. If the message does not

match, Bob must end all communication with the channel and delete all keys and values. Have a message that does not match could be an indicator of an attack or a compromise in the system. But if the decrypted message is correct, Alice and Bob may use shared secret key K_{AB} for the entire duration of the communication session [43].

CHAPTER 4: ELECTRICAL DESIGN AND RESULTS

Diabetes Mellitus is a disease that occurs when a person is unable to naturally control the amount of glucose sugar in their blood. It is caused by a lack of insulin either due to an inability to produce the cells or insulin resistance. The symptoms can include fatigue, weight loss, weight gain, hunger, dehydration and vision loss. If left untreated it can lead to heart attack, stroke, kidney failure, and peripheral nerve damage. Approximately 1.25 million American children suffer from diabetes. Also, diabetes was seventh on the leading causes of death in the United States[6].

Currently diabetes is treated with an insulin infusion pump, dieting, change of lifestyle and medication. In order to properly administer the required dosage of medication, it is essential for a patient to continuously monitor their blood glucose. Continuous monitoring of ones glucose can help a doctor and their patient better understand the correct dose of insulin or other modifications to treat the disease. Everyday more and more devices seem to become wireless. There are already commercially available insulin pumps and glucose monitors that can be purchased from any pharmacy[7]. Although these devices can be helpful, encryption is not always a top priority. It is easy to imagine how detrimental it would be if a hacker changed blood glucose readings or insulin dosages wirelessly. Hospital could also eventually be compromised as an increasing amount of infrastructure relies on an internet connection. These include patient records, infusion pump dosages and other sensitive data[1]. Although I am encrypting a blood glucose reading in this project, I believe this method can be applied to any biomedical device.

4.1 Glucose Test Strips

Most commercially available glucose strips use an electroenzymatic reaction in order to produce the blood glucose reading. The presence of glucose oxidase catalyzes a reaction between glucose with oxygen causing an increase in pH. This causes the production of Gluconicacid and hydrogen peroxide. The test strip measures changes in one or several of these components to determine the concentration of glucose. The strips used in this design utilize three electrodes: Reference, Working and Trigger(counter).

Figure 6: Glucose Strip Electrodes, Source: M. Bindhammer, 2016

A negative voltage of -0.4 V is applied at the reference electrode. When blood is inserted into the strip, the previously mentioned chemical reaction occurs, generating a small current that is proportional to the glucose concentration. This happens because the blood acts as an electrolytic bridge between the reference and trigger electrode. After the chemical reaction stabilizes for five seconds, the voltage is read by an analogue to digital converter. The voltage is then converted into glucose concentration using a simple algebraic expression obtained from calibrating with a store bought glucometer and solutions with known glucose concentrations[8]. After plotting known glucose concentrations against transimpedance amp output voltage, it can be seen that their

relationship is almost completely linear. The formula $C_{Glucose} = 495.6 * V_{OUT} - 1275.5$ was used to calculate the final glucose concentration [10].

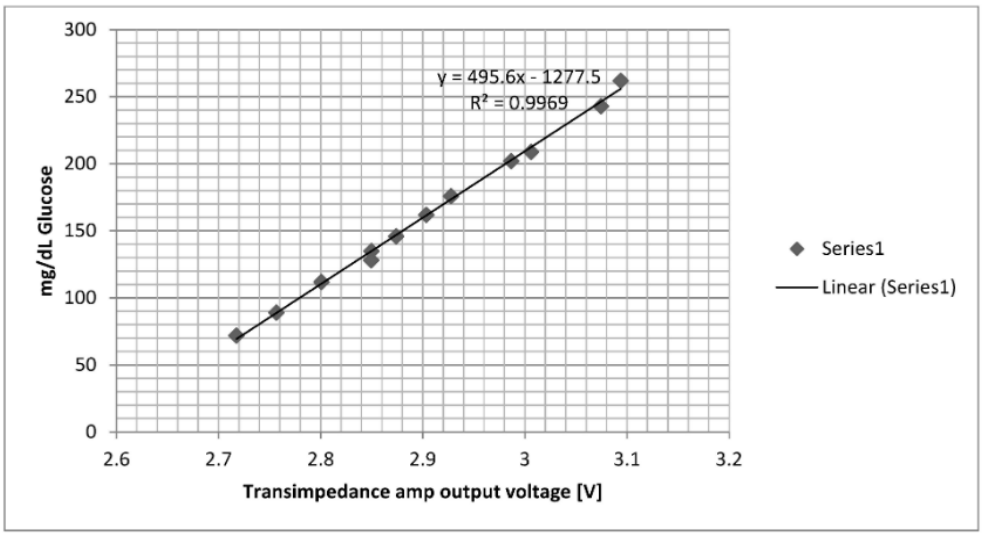

Figure 7: Relation between the glucose concentration and the amp output voltage is nearly linear, Source: M. Bindhammer, 2016

4.2 Design

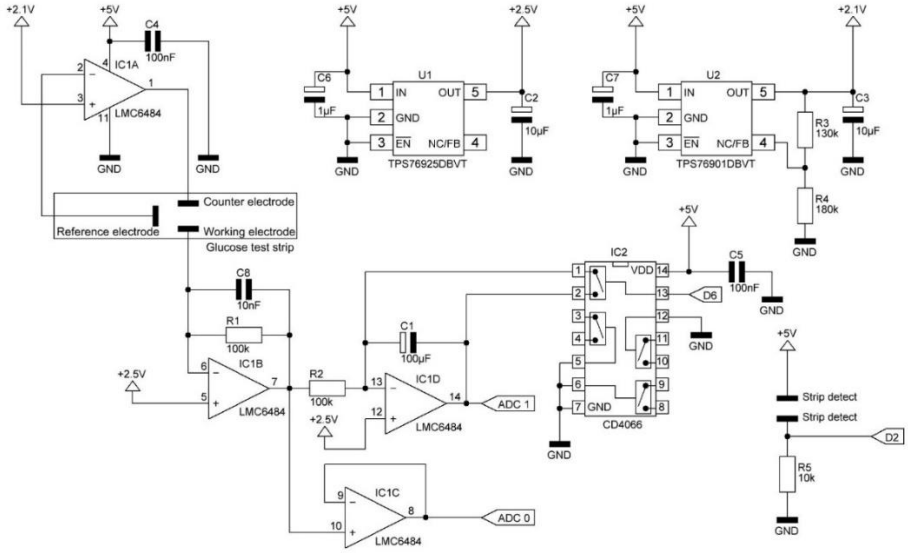

Figure 8: Glucose Meter Schematic, Source: M. Bindhammer, 2016

Above is the schematic used for the fabrication of the glucose meter. It is based off a design created by scientist M. Bindhammer in Germany. The low dropout regulator U1 provides a fixed output of 2.5V. Integration involves a known start time and end time, a reset circuit must be included to establish the start time before each integration time period. The integration end time occurs when the measurement is read. The IC2 simply short-circuits the capacitor C1 if digital output D6 goes HIGH. IC1A and IC1C are configured as unity gain buffer amplifiers. In this configuration, the entire output voltage is fed back into the inverting input. The difference between the non-inverting input voltage and the inverting input voltage is amplified by the op-amp. This connection forces the op-amp to adjust its output voltage simply equal to the input voltage. For IC1A this is only the case if an electrolytic bridge (blood) between the reference and counter electrode is applied. ONETOUCH ultra test strips were used in this experiment since they utilize the familiar three electrode design[10].

An Arduino Uno Rev3 was originally used as the microcontroller in this project. It has an operating voltage of 5Vwith 14 digital pins and 6 analogue pins that may be used as analogue to digital converters. This project required 2 digital pin inputs for glucose strip detection and 2 ADC’s for measuring current. When the glucose meter proved to be working, the microcontroller was switched to a Raspberry Pi Pico W. This is a much more capable microcontroller which is able to connect to the Internet or other Wi-Fi enabled devices.

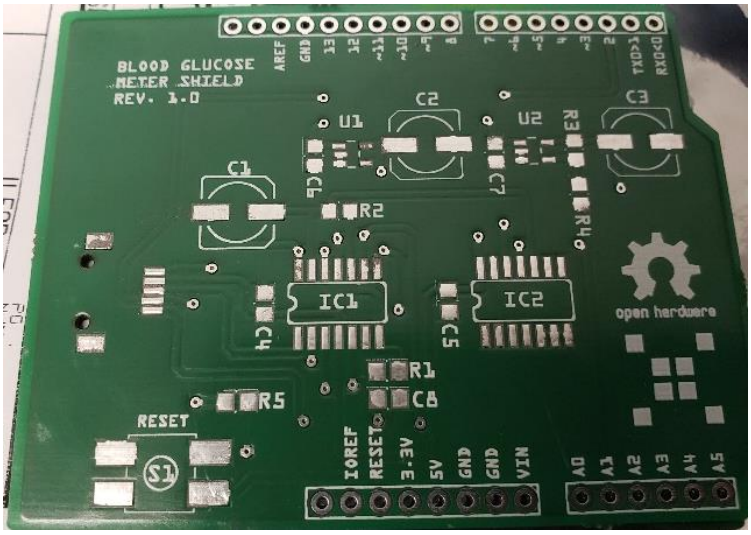

Figure 9: Unpopulated Glucose Meter PCB

Then I replaced the Arduino Uno with a Raspberry Pi (RPI) Pico W in order to perform more complex calculations and send data wirelessly. I then programmed the RPI Pico to produce a TCP server using C code. A serial console was used to observe outputs from the glucose meter. The key exchange protocol used in this experiment was X3DH. The Pico W that was connected to the glucose meter was programmed to behave as both Bob and the server. The server would generate a prekey bundle and send it to Alice. A one-time prekey was not utilized. The algorithm would also verify each Diffie-Hellman key. If they pass, Alice responds with the initial cyphertext and her public keys.

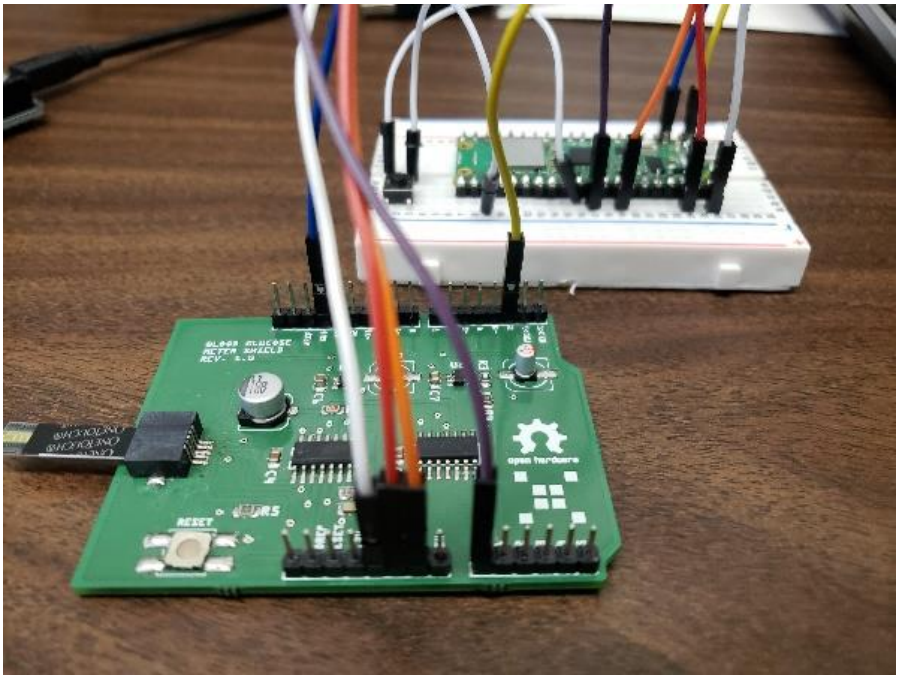

Figure 10: Populated PCB connected to Pico W

Finally, I programmed a TCP client socket on another Raspberry Pico W with hopes of receiving the encrypted data. The application will then decrypt the cipher text and produce the proper blood glucose reading. For the purposes of this experiment we used TinyJambu-128 for the encryption and decryption as this algorithm is optimized for lightweight applications. Unfortunately, the CRYSTALS-Kyber encryption algorithm did not run reliably on the performance restricted Pico W.

4.3 Results

After much experimentation, we were successfully able to develop a glucose monitoring system which could send the reading wirelessly to another device. Additionally, we were able to predictably decrypt the data on the client device by utilizing TinyJambu-128. The X3DH key exchange algorithm was also successfully

implemented into the system. The key exchange executed in 140,271 microseconds, with 17533875 clock cycles elapsed.

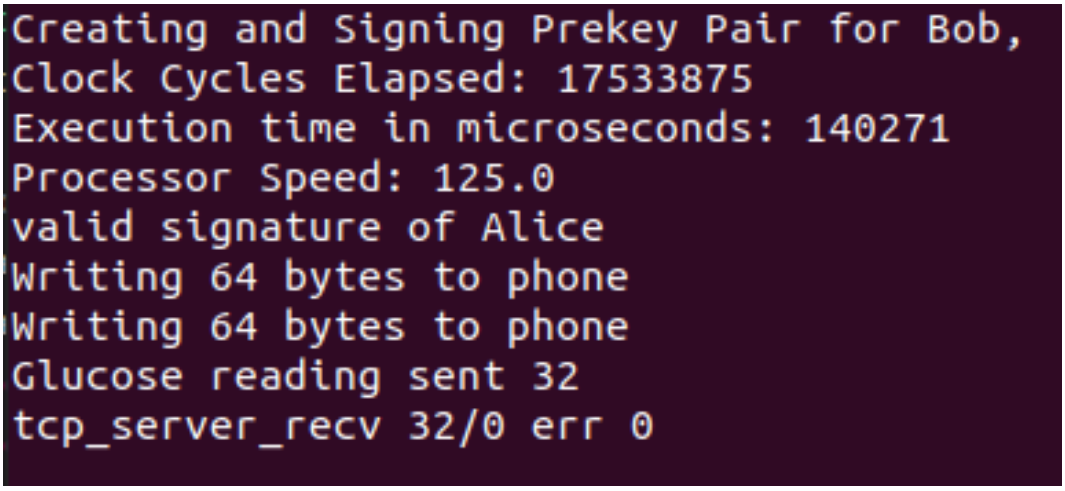

Figure 11: Speed test for X3DH on Pico W

CHAPTER 5: CONCLUSION

Cybersecurity will always be a challenge our society faces especially as our lives become ever more digital. Most people believe the worst thing a hacker can do is steal personal information such as a bank account or social security number. But as technology becomes more interlocked with healthcare and medicine, the implications become much more dire. Patients should be able to go to the hospital without fear of their information being breached or dosing being altered. These will become valid concerns as quantum computing continues to progress. This is why I believe this research is vital and the future of post quantum cryptography in biomedical applications should continue to be explored.

5.1 Future Work

Eventually I would like to incorporate a lattice based cryptosystem such as CRYSTALS-Kyber into the platform. This would require optimizing the algorithm to execute on lightweight devices. Possibly an existing hash function in the algorithm could be replaced with something more lightweight such as Ascon. Additionally, parallel or cloud computing could help with more complex operations such as polynomial multiplication.

REFERENCES

[1] Williams PA, Woodward AJ. Cybersecurity vulnerabilities in medical devices: a complex environment and multifaceted problem. Med Devices (Auckl). 2015 Jul 20;8:305-16. doi: 10.2147/MDER.S50048. PMID: 26229513; PMCID: PMC4516335.

[2] Bhowmik, Awnon & Menon, Unnikrishnan. (2020). Enhancing the NTRU Cryptosystem. International Journal of Computer Applications. 176. 46-53. 10.5120/ijca2020920320.

[3] Amundson, James & Sexton-Kennedy, Elizabeth. (2019). Quantum Computing. EPJ Web of Conferences. 214. 09010. 10.1051/epjconf/201921409010.

[4] Voas, Jeffrey & Bojanova, Irena. (2014). NIST: Building a solid foundation. ITProfessional. 16. 13-16. 10.1109/MITP.2014.21.

[5] Ahmad, Karim. “What Is Post-Quantum Cryptography & Why Is It Important?” MUO, 10 May 2022, https://www.makeuseof.com/what-is-post-quantum-cryptography.

[6] Centers for Disease Control and Prevention, “National Diabetes Statistics Report: Estimates of Diabetes and Its Burden in the United States”, 2014. Atlanta, GA: US Department of Health and Human Services; 2014.

[7] A.Tura et al., “Non-invasive glucose monitoring: Assessment of technologies and devices according to quantitative criteria,” Diabetes Res. Clinical Practice, vol. 77, no. 1, pp. 16–40, Jul. 2007.

[8] Yanez, Miriam. “Glucose Meter Fundamentals and Design.” NXP Semiconductors, Jan. 2013, https://www.nxp.com/docs/en/application-note/AN4364.pdf.

[9] Jeffrey Hoffstein, Jill Pipher, Joseph H. Silverman: NTRU: A ring-based public key cryptosystem, Algorithmic Number Theory Symposium – ANTS 1998, Lecture Notes in Computer Science 1423, Springer-Verlag (1998), pages 267–288.

[10] Bindhammer, M. “Open Source Arduino Blood Glucose Meter Shield.” Hackaday.io, 2016, https://hackaday.io/project/11719/logs.

[11] Rahimi Moosavi, Sanaz & Nigussie, Ethiopia & Levorato, Marco & Virtanen, Seppo & Isoaho, Jouni. (2018). Performance Analysis of End-to-End Security Schemes in Healthcare IoT. Procedia Computer Science. 130. 432-439. 10.1016/j.procs.2018.04.064.

[12] K. Quist-Aphetsi and M. C. Xenya, "Securing Medical IoT Devices Using Diffie-Hellman and DES Cryptographic Schemes," 2019 International Conference on Cyber Security and Internet of Things (ICSIoT), 2019, pp. 105-108, doi: 10.1109/ICSIoT47925.2019.00025.

[13] J. Granjal et al. End-to-end transport-layer security for Internet-integrated sensing applications with mutual and delegated ECC public-key authentication. In International Conference on Networking, pages 1–9, 2013.

[14] C. Poon et al. A Novel Biometrics Method to Secure Wireless Body Area Sensor Networks for Telemedicine and m-Health. IEEE Communications Magazine, 44(4):73–81, 2006.

[15] S. R. Moosavi et al. Cryptographic key generation using ECG signal. In 14th IEEE Annual Consumer Communications Networking Conference (CCNC), pages 1024–1031, 2017.

[16] Shrimali, Rahil. “How IOT Is Transforming the Healthcare Industry.” Embedded Computing Design, 30 June 2020, https://embeddedcomputing.com/application/healthcare/telehealth-healthcare-iot/how-iot-is-transforming-the-healthcare-industry.

[17] Chacko, Anil & Hayajneh, Thaier. (2018). Security and Privacy Issues with IoT in Healthcare. EAI Endorsed Transactions on Pervasive Health and Technology. 4. 155079. 10.4108/eai.13-7-2018.155079.

[18] G. H. Zhang, C. C. Y. Poon, and Y. T. Zhang, ‘‘A fast key generation method based on dynamic biometrics to secure wireless body sensor networks for p-health,’’ in Proc. Annu. Int. Conf. IEEE Eng. Med. Biol., Aug./Sep. 2010, pp. 2034–2036.

[19] T. William, “Healthcare’s ‘Internet of Things’ should be the ‘security of Things’,” 19 May 2015.

[20] BCC Research, “Mobile health (mHealth) Technologies and global markets 2014” 14 March 2014.

[21]“Body Guardian Mini Plus.” Preventice Solutions, 2019, https://www.preventicesolutions.com/patients/body-guardian-mini-plus.

[22] T. Harriet, “How the ‘Internet of Things’ could be fatal,” 4 March 2016; http://www.cnbc.com/2016/03/04/howthe-internet-of-things-could-be-fatal.html

[23] C. Catalin, “Thousands of IoT Medical Devices Found Vulnerable to Online Attacks,” 29 September 2015; http://news.softpedia.com/news/thousands-of-iotmedical-devices-found-vulnerable-to-online-attacks- 493144.shtml

[24]“Securing Wireless Infusion Pumps.” NCCoE, NIST, 2020, https://www.nccoe.nist.gov/healthcare/securing-wireless-infusion-pumps.

[25] S. Mahmood, “Medjacking: The newest health care risk?” 24 September 2015; http://www.healthcareitnews.com/news/medjackingnewest-healthcare-risk

[26]K. Lee, “Healthcare IoT security issues: Risks and what to do about them,” December 2015; http://searchhealthit.techtarget.com/feature/HealthcareIoT-security-issues-Risks-and-what-to-do-about-them

[27] Kumar, Adarsh & Ottaviani, Carlo & Gill, Sukhpal Singh & Buyya, Rajkumar. (2021). Securing the future internet of things with post-quantum cryptography.

[28] Agilepq Q3 2019 Report, A Guide to Post-Quantum Security for IoT Devices, URL:https://agilepq.com/wp-content/uploads/2020/02/Post_Quantum_IoT_WP.pdf

[29]Palmieri, P., 2018, May. Hash-based signatures for the internet of things: position paper. In Proceedings of the 15th ACM International Conference on Computing Frontiers (pp. 332-335).

[30] Xu, R., Cheng, C., Qin, Y. and Jiang, T., 2018. Lighting the way to a smart world: lattice-based cryptography for internet of things. arXiv preprint arXiv:1805.04880.

[31]Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. pages 284–293, 1997.

[32] Zvika Brakerski, Adeline Langlois, Chris Peikert, Oded Regev, and Damien Stehlé. Classical hardness of learning with errors. Pages 575–584, 2013.

[33] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. pages 595–618, 2009

[34] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018.

[35] A. Bogdanov, F. Mendel, F. Regazzoni, V. Rijmen, and E. Tischhauser. ALE: AES-Based Lightweight Authenticated Encryption. In Fast Software Encryption, 2013.

[36] J. Borghoff, A. Canteaut, T. G¨uneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. Thomsen, and T. Yal¸cn. PRINCE – A Low-Latency Block Cipher for Pervasive Computing Applications. In X. Wang and K. Sako, editors, Advances in Cryptology – ASIACRYPT 2012, volume 7658 of Lecture Notes in Computer Science, pages 208–225. Springer Berlin Heidelberg, 2012.

[37] Wu, Hongjun and Tao Huang. “TinyJAMBU : A Family of Lightweight Authenticated Encryption Algorithms (Version 2).” (2019).

Boaz Barak and Shai Halevi. A model and architecture for pseudo-random

generation with applications to /dev/random. Cryptology ePrint Archive, Report 2005/029, 2005. https://eprint.iacr.org/2005/029.

[38] Mihir Bellare and Phillip Rogaway. Entity Authentication and Key Distribution. In Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '93, page 232–249, Berlin, Heidelberg, 1993. Springer-Verlag.

[39] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Proceedings of the 1st ACM conference on Computer and communications security, pages 62–73, 1993.

[40] Mihir Bellare and Phillip Rogaway. Code-Based Game-Playing Proofs and the Security of Triple Encryption. Cryptology ePrint Archive, Report 2004/331, 2004. https://eprint.iacr.org/2004/331.

[41] Dan Boneh. The decision diffie-hellman problem. In International Algorithmic Number Theory Symposium, pages 48–63. Springer, 1998.

[42] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In International Conference on the Theory and Applications of Cryptographic Techniques, pages 255–271. Springer, 2003.

[43]Have, Ferran van der. "The x3DH Protocol: A Proof of Security." Radboud University, Department of Computer Science, 22 Jan. 2022, https://www.cs.ru.nl/bachelors-theses/

APPENDIX A

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"
#include "lwip/pbuf.h"
#include "lwip/tcp.h"
#include "hardware/gpio.h"
#include "hardware/adc.h"
#include "math.h"
#include "hardware/pio.h"
#include "bob/crypto_aead.h"
#include "../X3DH/ed25519/src/ed25519.h"
#include "../X3DH/sha/rfc6234/sha.h"

#define TCP_PORT 4242
#define DEBUG_printf printf
#define BUF_SIZE 64
#define TEST_ITERATIONS 4
#define POLL_TIME_S 10

int FetchPreKeyBundle(unsigned char *bob_id_public_key, unsigned char
*bob_spk_public_key, unsigned char *bob_spk_signature, int message_type){};
void get_dh_output(unsigned char *bob_id_public_key, unsigned char
*ephemeral_private_key, unsigned char *id_private_key,
        unsigned char *bob_spk_public_key, unsigned char *dh_final);
void get_shared_key(unsigned char *dh_final, SHAversion whichSha, const unsigned
char *salt_len, const unsigned char *info,
```

```
                unsigned char *output_key, int okm_len);
void encrypt(const unsigned char *m, unsigned long long mlen, const unsigned char *k,
char* c, uint64_t* c_length);


void decrypt(const unsigned char *c, const unsigned char *m, unsigned long long clen,
const unsigned char *k, uint64_t* m_length);


typedef struct TCP_SERVER_T_ {
    struct tcp_pcb *server_pcb;
    struct tcp_pcb *client_pcb;
    bool complete;
    unsigned char buffer_sent[BUF_SIZE];
    unsigned char conn[8];
    char buffer_recv[BUF_SIZE];
    int sent_len;
    int recv_len;
    int run_count;
    unsigned char bob_id_public_key[32]; //Bob's Identity Public Key
    unsigned char bob_id_private_key[64];//Bob's Identity Private Key
    unsigned char bob_seed[32]; //Seed to generate new keys
    unsigned char bob_spk_public_key[32]; //Bob's Signed prekey public
    unsigned char bob_spk_private_key[64]; //Bob's Signed prekey private
    unsigned char alice_id_public_key[32]; //Alice's Identity Public Key
    unsigned char alice_ephemeral_public_key[32];
    unsigned char bob_spk_signature[64];
    unsigned char dh1_bob[32];
    unsigned char dh2_bob[32];
    unsigned char dh3_bob[32];
    unsigned char dh_final_bob[96];
```

```
    unsigned char hex_hkdf_output_bob[128];
    unsigned char cipher[32];
    } TCP_SERVER_T;
static TCP_SERVER_T* tcp_server_init(void) {
    TCP_SERVER_T *state = calloc(1, sizeof(TCP_SERVER_T));
    if (!state) {
        DEBUG_printf("failed to allocate state\n");
        return NULL;
    }
    return state;
}
static err_t tcp_server_close(void *arg) {
    TCP_SERVER_T *state = (TCP_SERVER_T*)arg;
    err_t err = ERR_OK;
    if (state->client_pcb != NULL) {
        tcp_arg(state->client_pcb, NULL);
        tcp_poll(state->client_pcb, NULL, 0);
        tcp_sent(state->client_pcb, NULL);
        tcp_recv(state->client_pcb, NULL);
        tcp_err(state->client_pcb, NULL);
        err = tcp_close(state->client_pcb);
        if (err != ERR_OK) {
            DEBUG_printf("close failed %d, calling abort\n", err);
            tcp_abort(state->client_pcb);
            err = ERR_ABRT;
        }
        state->client_pcb = NULL;
    }
```

```
    if (state->server_pcb) {
        tcp_arg(state->server_pcb, NULL);
        tcp_close(state->server_pcb);
        state->server_pcb = NULL;
    }
    return err;
}
static err_t tcp_server_result(void *arg, int status) {
    TCP_SERVER_T *state = (TCP_SERVER_T*)arg;
    if (status =! 0) {
        DEBUG_printf("test success\n");
    } else {
        DEBUG_printf("test failed %d\n", status);
    }
    state->complete = true;

   return tcp_server_close(arg);
}

static err_t tcp_server_sent(void *arg, struct tcp_pcb *tpcb, u16_t len) {
    TCP_SERVER_T *state = (TCP_SERVER_T*)arg;
    DEBUG_printf("Glucose reading sent %u\n", len);
    state->sent_len += len;

    if (state->sent_len >= BUF_SIZE)  // We should get the data back from the client
        state->recv_len = 0;
        DEBUG_printf("Waiting for response from glucose meter\n");
    }
```

```
  return ERR_OK;


}
err_t tcp_server_send_data(void *arg, struct tcp_pcb *tpcb)
{
    TCP_SERVER_T *state = (TCP_SERVER_T*)arg;
     if(state->run_count < 1)
    {


    ed25519_create_seed(state->bob_seed); //create randome seed
    ed25519_create_keypair(state->bob_id_public_key, state->bob_id_private_key, state-
>bob_seed);


//Generate SignedPreKey Pair for bob
    ed25519_create_seed(state->bob_seed); //create random seed
    ed25519_create_keypair(state->bob_spk_public_key, state->bob_spk_private_key,
state-> bob_seed);
    ed25519_sign(state->bob_spk_signature, state->bob_id_public_key, state-
>bob_id_private_key);


    if (ed25519_verify(state->bob_spk_signature, state-> bob_id_public_key)) {
        printf("\nvalid signature of ALice\n");
    } else {
        printf("\ninvalid signature\n");
        // Abort();
    }
   /* for (int i = 0; i < 64 ; i++)
    {
        state->bob_spk[i] = bob_spk_private_key[i];
```

```
    }*/

    }

   // unsigned char dh1_bob[32], dh2_bob[32], dh3_bob[32]; //DH exchanges - no opk so only 3 outputs


    //  BOB'S KEY EXCHANGES

    //DH1 = DH(IKA, SPKB)

    //ed25519_key_exchange(dh1_bob, alice_id_public_key, bob_spk_private_key);


    //DH2 = DH(EKA, IKB)

    //ed25519_key_exchange(dh2_bob, alice_ephemeral_public_key, bob_id_private_key);


    //DH3 = DH(EKA, SPKB)

    //ed25519_key_exchange(dh3_bob, alice_ephemeral_public_key, bob_spk_private_key);

    /*

    for(int i=0; i< BUF_SIZE; i++) {


    state->buffer_sent[i] =   //state->conn[i];


    }*/


    if (state->run_count == 0)

       {

          //send bob spk public key to client

          DEBUG_printf("Writing %ld bytes to phone\n", BUF_SIZE);

        err_t err = tcp_write(tpcb, state->bob_spk_public_key, 32, TCP_WRITE_FLAG_COPY);
```

```
        if (err != ERR_OK) {
        DEBUG_printf("Failed to write data %d\n", err);
        return tcp_server_result(arg, -1);}


        }


         if (state->run_count == 1)
       {   //send bob id public key
           DEBUG_printf("Writing %ld bytes to phone\n", BUF_SIZE);
        err_t err = tcp_write(tpcb, state -> bob_id_public_key,
32 ,TCP_WRITE_FLAG_COPY);
         if (err != ERR_OK) {
        DEBUG_printf("Failed to write data %d\n", err);
        return tcp_server_result(arg, -1);}


        }


         if (state->run_count == 2)
       {
           DEBUG_printf("Writing %ld bytes to phone\n", BUF_SIZE);
        err_t err = tcp_write(tpcb, state -> bob_id_public_key,
32 ,TCP_WRITE_FLAG_COPY);
         if (err != ERR_OK) {
        DEBUG_printf("Failed to write data %d\n", err);
        return tcp_server_result(arg, -1);}


        }
         if (state->run_count == 3)
       {
```

```
        DEBUG_printf("Writing %ld bytes to phone\n", BUF_SIZE);
      err_t err = tcp_write(tpcb, state -> cipher, 16 ,TCP_WRITE_FLAG_COPY);
       if (err != ERR_OK) {
      DEBUG_printf("Failed to write data %d\n", err);
      return tcp_server_result(arg, -1);}


      }
   state->sent_len = 0;


   // this method is callback from lwIP, so cyw43_arch_lwip_begin is not required, however you
   // can use this method to cause an assertion in debug mode, if this method is called when
   //cyw43_arch_lwip_begin IS needed


  // cyw43_arch_lwip_check();
   //err_t err = tcp_write(tpcb, state->buffer_sent, BUF_SIZE, TCP_WRITE_FLAG_COPY);


  /* if (err != ERR_OK) {
       DEBUG_printf("Failed to write data %d\n", err);
       return tcp_server_result(arg, -1);
   }*/
   return ERR_OK;


}


err_t tcp_server_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err) {
```

```
    TCP_SERVER_T *state = (TCP_SERVER_T*)arg;
   //if (!p) {
     //  return tcp_server_result(arg, -1);
    //}
    // this method is callback from lwIP, so cyw43_arch_lwip_begin is not required, however you
    // can use this method to cause an assertion in debug mode, if this method is called when
    // cyw43_arch_lwip_begin IS needed
    cyw43_arch_lwip_check();
    if (p->tot_len > 0) {
        DEBUG_printf("tcp_server_recv %d/%d err %d\n", p->tot_len, state->recv_len, err);


        // Receive the buffer
        const uint16_t buffer_left = BUF_SIZE - state->recv_len;
        state->recv_len += pbuf_copy_partial(p, state->buffer_recv + state->recv_len,
                                             p->tot_len > buffer_left ? buffer_left : p->tot_len, 0);
        tcp_recved(tpcb, p->tot_len);
    }
    pbuf_free(p);
    DEBUG_printf("\n");
    printf("\n");


    if(state->run_count == 0)
    {
          //DH1 = DH(IKA, SPKB)


          for (int i = 0; i < 32; i++)
```

```
        {
            state->alice_id_public_key[i] = state -> buffer_recv[i];
        }

            ed25519_key_exchange(state->dh1_bob, state->alice_id_public_key, state->bob_spk_private_key);
            printf("Exchanging DH1\n");
      }
    if(state->run_count == 1)
    {
          //DH1 = DH(IKA, SPKB)

          for (int i = 0; i < 32; i++)
        {
          state->alice_ephemeral_public_key[i] = state -> buffer_recv[i];
         }
            ed25519_key_exchange(state->dh2_bob, state->alice_ephemeral_public_key, state->bob_id_private_key);
            printf("Exchanging DH2\n");
    }

     if(state->run_count == 2)
    {
          //DH1 = DH(IKA, SPKB)

          ed25519_key_exchange(state->dh3_bob, state->alice_ephemeral_public_key, state->bob_spk_private_key);

            printf("Exchanging DH3\n");
```

```
        for(int j=0; j<96;j++)
    {
       if(j<32)state-> dh_final_bob[j] =state-> dh1_bob[j];
       if(j>=32 && j< 64) state-> dh_final_bob[j] = state->dh2_bob[j%32];
       if(j>=64)  state->dh_final_bob[j] = state->dh3_bob[j%32];
    }
   get_shared_key(state->dh_final_bob, SHA512, NULL, NULL, state-
>hex_hkdf_output_bob, 128);
     for(int i=0; i<128;i++)
    {
       // if (i%16 == 0) printf("\t");
       printf("%d\n",state -> hex_hkdf_output_bob[i]);


    }
    printf("end\n");


     const size_t SIZE_OF_END = 0;
   // const size_t BUFFER_SIZE = ((SAMPLE_FREQUENCY *
BYTES_PER_SAMPLE) / 2);
    const size_t BUFFER_SIZE = 8;
    // below stores 8, 32-bit values
    // const size_t BUFFER_SIZE = ((BYTES_PER_SAMPLE) * 8);


    // const size_t SAMPLE_ARR_SIZE = BUFFER_SIZE + SIZE_OF_END;
    const size_t SAMPLE_ARR_SIZE = BUFFER_SIZE + SIZE_OF_END;
     unsigned long long c_length = 16;                                  // ciphertext length
    unsigned char c[c_length];


    //unsigned long long m_length = 24;
```

```
//unsigned char m[m_length];


unsigned char sampleArr[SAMPLE_ARR_SIZE];
for(int i=0; i<8;i++)
{
   sampleArr[i] = state->conn[i];
}
 unsigned char k[16];


for (int i = 0; i < sizeof(k); i++)
{
   k[i] = state->hex_hkdf_output_bob[i];
}
encrypt(sampleArr, SAMPLE_ARR_SIZE, state -> hex_hkdf_output_bob, c,
&c_length)
for(int i=0; i< 32;i++)
{
  state ->cipher[i] = c[i];
}
}
// Have we have received the whole buffer
if (state->recv_len == BUF_SIZE || state->recv_len != BUF_SIZE) {


    // check it matches
   //  if (memcmp(state->buffer_sent, state->buffer_recv, BUF_SIZE) != 0) {
       // DEBUG_printf("buffer mismatch\n");
        //return tcp_server_result(arg, -1);
    //}
    DEBUG_printf("tcp_server_recv buffer ok\n");
```

```
        // Test complete?
        state->run_count++;
        if (state->run_count >= TEST_ITERATIONS) {
            tcp_server_result(arg, 0);
            return ERR_OK;
        }

    DEBUG_printf("\n");
        // Send another buffer
       return tcp_server_send_data(arg, state->client_pcb);
    }
   return ERR_OK;
}

static err_t tcp_server_poll(void *arg, struct tcp_pcb *tpcb) {
    DEBUG_printf("tcp_server_poll_fn\n");
    return tcp_server_result(arg, -1); // no response is an error?
}

static void tcp_server_err(void *arg, err_t err) {
    if (err != ERR_ABRT) {
        DEBUG_printf("tcp_client_err_fn %d\n", err);
       tcp_server_result(arg, err);
    }
}

static err_t tcp_server_accept(void *arg, struct tcp_pcb *client_pcb, err_t err) {
```

```
    TCP_SERVER_T *state = (TCP_SERVER_T*)arg;
    if (err != ERR_OK || client_pcb == NULL) {
        DEBUG_printf("Failure in accept\n");
       tcp_server_result(arg, err);
        return ERR_VAL;
    }
    DEBUG_printf("Smartphone connected\n");

    state->client_pcb = client_pcb;
    tcp_arg(client_pcb, state);
    tcp_sent(client_pcb, tcp_server_sent);
    tcp_recv(client_pcb, tcp_server_recv);
    tcp_poll(client_pcb, tcp_server_poll, POLL_TIME_S * 2);
    tcp_err(client_pcb, tcp_server_err);

    return tcp_server_send_data(arg, state->client_pcb);
}

static bool tcp_server_open(void *arg) {
    TCP_SERVER_T *state = (TCP_SERVER_T*)arg;
    DEBUG_printf("Starting server at %s on port %u\n",
ip4addr_ntoa(netif_ip4_addr(netif_list)), TCP_PORT);

    struct tcp_pcb *pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
    if (!pcb) {
        DEBUG_printf("failed to create pcb\n");
        return false;
    }
```

```
    err_t err = tcp_bind(pcb, NULL, TCP_PORT);
    if (err) {
        DEBUG_printf("failed to bind to port %d\n");
        return false;
    }

    state->server_pcb = tcp_listen_with_backlog(pcb, 1);
    if (!state->server_pcb) {
        DEBUG_printf("failed to listen\n");
        if (pcb) {
            tcp_close(pcb);
        }
        return false;
    }

    tcp_arg(state->server_pcb, state);
    tcp_accept(state->server_pcb, tcp_server_accept);

    return true;
}

void run_tcp_server_test(int conn) {
    TCP_SERVER_T *state = tcp_server_init();
int n = log10(conn) + 1;
    int i;

    for (i = n-1; i >= 0; --i, conn /= 10)
    {
```

```
        state->conn[i] = (conn % 10) + '0';
    }
    state->conn[3] = 'm';
    state->conn[4] = 'g';
    state->conn[5] = '/';
    state->conn[6] = 'd';
    state->conn[7] = 'L';
    if (!state) {
        return;
    }
    if (!tcp_server_open(state)) {
        tcp_server_result(state, -1);
        return;
    }
    while(!state->complete) {

        // the following #ifdef is only here so this same example can be used in multiple modes;
        // you do not need it in your code
#if PICO_CYW43_ARCH_POLL
        // if you are using pico_cyw43_arch_poll, then you must poll periodically from your
        // main loop (not from a timer) to check for WiFi driver or lwIP work that needs to be done.
        cyw43_arch_poll();
        sleep_ms(1);
#else
        // if you are not using pico_cyw43_arch_poll, then WiFI driver and lwIP work
        // is done via interrupt in the background. This sleep is just an example of some (blocking)
```

```
        // work you might be doing.
        sleep_ms(1000);
#endif
    }
    free(state);
}

// Function to compute `a^m mod n`
int compute(int a, int m, int n)
{
    int r;
    int y = 1;

    while (m > 0)
    {
        r = m % 2;

        // fast exponention
        if (r == 1) {
            y = (y*a) % n;
        }
        a = a*a % n;
        m = m / 2;
    }

    return y;
}
```

```
int main() {
    char ssid[] = "TP-Link_A15C";
    char pass[] = "43193455";
    stdio_init_all();

   const uint SENSOR_PIN = 0;

// measurement starts if transimpedance amp output voltage > threshold
   float threshold = 2.8;
    adc_init();
    gpio_init(SENSOR_PIN);
    gpio_set_dir(SENSOR_PIN, GPIO_IN);

    // Make sure GPIO is high-impedance, no pullups etc
    adc_gpio_init(26);
    // Select ADC input 0 (GPIO26)
    adc_select_input(0);

    while(1){

    if(gpio_get(SENSOR_PIN) != 0) break;
    }

    printf("APPLY BLOOD\n");

    float current_voltage;
    while(1) {
    current_voltage = adc_read() * (3.3f / (1 << 12));
```

```
    if(current_voltage > threshold) break;
    }
    printf("Reading will be ready in 5 seconds\n");
    // count down timer
  for(int i = 5; i > 0; i--) {
   sleep_ms(1000);
    printf("%d",i);
    if(i > 1) printf(", \n");
    else printf("\n");


  }
 current_voltage = adc_read() * (3.3f / (1 << 12));
   // compute concentration
  current_voltage = adc_read() * (3.3f / (1 << 12));


  float conn = 495.6 * current_voltage - 1266 + 150 + 1000 + 300;


printf("Check phone for results\n");
  //unit8_t concentration = conn;
  printf("%f mg/dL\n", conn);
int con = conn;
//     int i =
10;:picow_tcp_server.c:((.text.tcp_server_recv.text.tcp_server_recv+0x+0x216)216:
undefined reference to `)crypto_aead_encrypt: u
//  while (i--) {
//    printf("Countdown %i\n", i);
//    sleep_ms(1000);
//  }
```

```
    if (cyw43_arch_init()) {
        printf("failed to initialise\n");
        return 1;
    }
    cyw43_arch_enable_sta_mode();

    //printf("Connecting to WiFi...\n");
    if (cyw43_arch_wifi_connect_timeout_ms(ssid, pass,
CYW43_AUTH_WPA2_AES_PSK, 50000)) {
        printf("failed to connect.\n");
        return 1;
    } else {
        printf("Connected.\n");
    }
    run_tcp_server_test(con);

    cyw43_arch_deinit();
    return 0;
    }

void encrypt(const unsigned char *m, unsigned long long mlen, const unsigned char *k,
char* c, uint64_t* c_length)
{
    // unsigned long long c_length = 80;                                    //
ciphertext length
    // unsigned char c[c_length];                                          //
ciphertext
    unsigned long long *clen = c_length;                                    //
ciphertext length pointer
    const unsigned char ad[] = {0x00};                                      //
associated data
```

```
    unsigned long long adlen = sizeof(ad);                                  //
associated data length
    const unsigned char *nsec;                                              // secret
message number
    const unsigned char npub[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0A, 0x0B}; // public message number


    crypto_aead_encrypt(c, clen, m, mlen, ad, adlen, nsec, npub, k);


    printf("Ciphertext = ");
    for (int i = 0; i < *c_length; i++)
    {
        printf("%02X|", c[i]);
    }
    printf("\n");
}


void get_dh_output(unsigned char *bob_id_public_key, unsigned char
*alice_ephemeral_private_key, unsigned char *alice_id_private_key,
                   unsigned char *bob_spk_public_key, unsigned char *dh_final)
{
    // DH outputs
    unsigned char dh1[32], dh2[32], dh3[32]; // DH exchanges - no opk so only 3 outputs


    ed25519_key_exchange(dh1, bob_spk_public_key, alice_id_private_key);


    // DH2 = DH(EKA, IKB)
    ed25519_key_exchange(dh2, bob_id_public_key, alice_ephemeral_private_key);


    // DH3 = DH(EKA, SPKB)
```

```
    ed25519_key_exchange(dh3, bob_spk_public_key, alice_ephemeral_private_key);


    // Concatenating dh outputs - because strcat, strncat and memcpy doesn't seem to work.
    for (int j = 0; j < 96; j++)
    {
        if (j < 32)
            dh_final[j] = dh1[j];
        if (j >= 32 && j < 64)
            dh_final[j] = dh2[j % 32];
        if (j >= 64)
            dh_final[j] = dh3[j % 32];
    }
}


void get_shared_key(unsigned char *dh_final, SHAversion whichSha, const unsigned
char *salt, const unsigned char *info,
                    unsigned char *output_key, int okm_len)
{
    int salt_len; // The length of the salt value (a non-secret random value) (ignored if
SALT==NULL)
    int info_len; // The length of optional context and application (ignored if info==NULL)
    int ikm_len;  // The length of the input keying material
    uint8_t okm_integer[okm_len];
    ikm_len = 96;


    if (salt == NULL)
        salt_len = 0;
    if (info == NULL)
        info_len = 0;
```

```
    if (hkdf(whichSha, salt, salt_len, dh_final, ikm_len, info, info_len, okm_integer,
okm_len) == 0)
    {
        printf("HKDF (shared secret):\n");
    }
    else
    {
        fprintf(stderr, "HKDF is invalid\n");
    }

    for (int i = 0; i < okm_len; i++)
    {
        output_key[i] = okm_integer[i];
        //printf("%d", output_key[i]);
    }
}

void decrypt(const unsigned char *c, const unsigned char *m, unsigned long long clen,
const unsigned char *k, uint64_t* m_length)
{
    unsigned long long *mlen = m_length;                                    //
plaintext length pointer
    const unsigned char ad[] = {0x00};                                      //
associated data
    unsigned long long adlen = sizeof(ad);                                  //
associated data length
    unsigned char *nsec;                                                    // secret
message number
    const unsigned char npub[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0A, 0x0B}; // public message number
```

```
    crypto_aead_decrypt(m, mlen, nsec, c, clen, ad, adlen, npub, k);
    // print
    printf("Plaintext = ");
    for (int i = 0; i < *m_length; i++)
    {
        printf("%02X|", m[i]);
    }
    printf("\n");
}
```

APPENDIX B

```
#include <string.h>
#include <time.h>
#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"
#include "lwip/pbuf.h"
#include "lwip/tcp.h"
#include <stdio.h>
#include "../X3DH/ed25519/src/ed25519.h"
#include "../X3DH/sha/rfc6234/sha.h"
#include "include/tinyJambu/crypto_aead.h"

#if !defined(TEST_TCP_SERVER_IP)
#error TEST_TCP_SERVER_IP not defined
#endif

#define TCP_PORT 4242
#define DEBUG_printf printf
#define BUF_SIZE 64

#define TEST_ITERATIONS 4
#define POLL_TIME_S 10

#if 0
static void dump_bytes(const uint8_t *bptr, uint32_t len) {
    unsigned int i = 0;

    printf("dump_bytes %d", len);
    for (i = 0; i < len;) {
```

```
        if ((i & 0x0f) == 0) {
            printf("\n");
        } else if ((i & 0x07) == 0) {
            printf(" ");
        }
        printf("%02x ", bptr[i++]);
    }
    printf("\n");
}
#define DUMP_BYTES dump_bytes
#else
#define DUMP_BYTES(A,B)
#endif

typedef struct TCP_CLIENT_T_ {
    struct tcp_pcb *tcp_pcb;
    ip_addr_t remote_addr;
    uint8_t buffer[BUF_SIZE];
    int buffer_len;
    int sent_len;
    bool complete;
    int run_count;
    bool connected;
    unsigned char alice_id_public_key[32]; //Bob's Identity Public Key
    unsigned char alice_id_private_key[64]; //Bob's Identity Private Key
    unsigned char alice_seed[32]; //Seed to generate new keys
    unsigned char alice_ephemeral_public_key[32]; //Alice public identity key
    unsigned char alice_ephemeral_private_key[64]; //Alice ephemeral/generated key
```

```
    unsigned char bob_spk_public_key[32];
    unsigned char bob_id_public_key[32]; //Bob's Signed prekey public
    unsigned char dh1_alice[32];
    unsigned char dh2_alice[32];
    unsigned char dh3_alice[32];
    unsigned char dh_final_alice[96];
    unsigned char hex_hkdf_output_alice[128];
    unsigned char cipher[32];

} TCP_CLIENT_T;

static err_t tcp_client_close(void *arg) {
    TCP_CLIENT_T *state = (TCP_CLIENT_T*)arg;
    err_t err = ERR_OK;
    if (state->tcp_pcb != NULL) {
        tcp_arg(state->tcp_pcb, NULL);
        tcp_poll(state->tcp_pcb, NULL, 0);
        tcp_sent(state->tcp_pcb, NULL);
        tcp_recv(state->tcp_pcb, NULL);
        tcp_err(state->tcp_pcb, NULL);
        err = tcp_close(state->tcp_pcb);
        if (err != ERR_OK) {
            DEBUG_printf("close failed %d, calling abort\n", err);
            tcp_abort(state->tcp_pcb);
            err = ERR_ABRT;
        }
        state->tcp_pcb = NULL;
    }
```

```
    return err;
}

void decrypt(const unsigned char* c, unsigned long long clen, const unsigned char* k);
void encrypt(const unsigned char* m, unsigned long long mlen, const unsigned char* k);

// Called with results of operation
static err_t tcp_result(void *arg, int status) {
    TCP_CLIENT_T *state = (TCP_CLIENT_T*)arg;
    if (status == 0) {
        DEBUG_printf("test success\n");
    } else {
        DEBUG_printf("test failed %d\n", status);
    }
    state->complete = true;
    return tcp_client_close(arg);
}

static err_t tcp_client_sent(void *arg, struct tcp_pcb *tpcb, u16_t len) {
    TCP_CLIENT_T *state = (TCP_CLIENT_T*)arg;
    DEBUG_printf("tcp_client_sent %u\n", len);
    state->sent_len += len;

    if (state->sent_len <= BUF_SIZE || state->sent_len > BUF_SIZE) {//->buffer

        state->run_count++;
        if (state->run_count >= TEST_ITERATIONS) {
            tcp_result(arg, 0);
```

```
            return ERR_OK;
        }

        // We should receive a new buffer from the server
        state->buffer_len = 0;
        state->sent_len = 0;
        DEBUG_printf("Waiting for buffer from server\n");
    }

    return ERR_OK;
}

static err_t tcp_client_connected(void *arg, struct tcp_pcb *tpcb, err_t err) {
    TCP_CLIENT_T *state = (TCP_CLIENT_T*)arg;
    if (err != ERR_OK) {
        printf("connect failed %d\n", err);
        return tcp_result(arg, err);
    }
    state->connected = true;
    DEBUG_printf("Waiting for buffer from server\n");
    return ERR_OK;
}

static err_t tcp_client_poll(void *arg, struct tcp_pcb *tpcb) {
    DEBUG_printf("tcp_client_poll\n");
    return tcp_result(arg, -1); // no response is an error?
}
```

```
static void tcp_client_err(void *arg, err_t err) {
    if (err != ERR_ABRT) {
        DEBUG_printf("tcp_client_err %d\n", err);
        tcp_result(arg, err);
    }
}


err_t tcp_client_recv(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t err) {
 //Defining parameters for Alice


    TCP_CLIENT_T *state = (TCP_CLIENT_T*)arg;
    if (!p) {
        return tcp_result(arg, -1);
    }
    // this method is callback from lwIP, so cyw43_arch_lwip_begin is not required,
however you
    // can use this method to cause an assertion in debug mode, if this method is called
when
    // cyw43_arch_lwip_begin IS needed
    cyw43_arch_lwip_check();
    if (p->tot_len > 0) {
        DEBUG_printf("recv %d err %d\n", p->tot_len, err);
        for (struct pbuf *q = p; q != NULL; q = q->next) {
            DUMP_BYTES(q->payload, q->len);
        }
        // Receive the buffer
        const uint16_t buffer_left = BUF_SIZE - state->buffer_len;
        state->buffer_len += pbuf_copy_partial(p, state->buffer + state->buffer_len,
                                               p->tot_len > buffer_left ? buffer_left : p->tot_len, 0);
```

```
        tcp_recved(tpcb, p->tot_len);
    }
    pbuf_free(p);
if(state->run_count < 1)
{
            for (int i = 0; i < 32; i++)
    {
        state->bob_spk_public_key[i] = state->buffer[i];
    }


     //Verifying on Alice's side
    ed25519_create_seed(state->alice_seed);
    ed25519_create_keypair(state->alice_id_public_key, state-> alice_id_private_key,
state-> alice_seed);


    //Generate Ephemeral keys
    ed25519_create_seed(state->alice_seed);
    ed25519_create_keypair(state->alice_ephemeral_public_key,state->
alice_ephemeral_private_key, state->alice_seed);




}
    // If we have received the whole buffer, send it back to the server
    if (state->buffer_len){// == BUF_SIZE) {
       // DEBUG_printf("Writing %d bytes to server\n", state->buffer_len);
        //err_t err = tcp_write(tpcb, state->buffer, state->buffer_len,
TCP_WRITE_FLAG_COPY);


       if (state->run_count == 0)
```

```
      {


        DEBUG_printf("Writing %d bytes to server\n", state->buffer_len);
       err_t err = tcp_write(tpcb, state->alice_id_public_key, state->buffer_len,TCP_WRITE_FLAG_COPY);


        ed25519_key_exchange(state->dh1_alice, state->bob_spk_public_key, state->alice_id_private_key);
    printf("Exchanging DH1\n");


       }
        if (state->run_count == 1)
      {
        DEBUG_printf("Writing %d bytes to server\n", state->buffer_len);
       err_t err = tcp_write(tpcb, state -> alice_ephemeral_public_key, state->buffer_len,TCP_WRITE_FLAG_COPY);


       for (int i = 0; i < 32; i++)
    {
       state->bob_id_public_key[i] = state->buffer[i];
    }


       ed25519_key_exchange(state->dh2_alice, state->bob_id_public_key, state->alice_ephemeral_private_key);
    printf("Exchanging DH2\n");



       }


        if (state->run_count == 2)
```

```
      {


       DEBUG_printf("Writing %d bytes to server\n", state->buffer_len);

      err_t err = tcp_write(tpcb, state -> alice_ephemeral_public_key, state->buffer_len,TCP_WRITE_FLAG_COPY);

       ed25519_key_exchange(state-> dh3_alice, state-> bob_spk_public_key, state->alice_ephemeral_private_key);


       printf("Exchanging DH3\n");


          for(int j=0; j<96;j++)
   {
      if(j<32) state->dh_final_alice[j] = state->dh1_alice[j];
      if(j>=32 && j< 64)  state->dh_final_alice[j] = state->dh2_alice[j%32];
      if(j>=64)  state->dh_final_alice[j] = state->dh3_alice[j%32];
   }
    get_shared_key(state -> dh_final_alice, SHA512, NULL, NULL, state -> hex_hkdf_output_alice, 128);
       for(int i=0; i<16;i++)
   {
      // if (i%16 == 0) printf("\t");
      printf("%02X\n",state -> hex_hkdf_output_alice[i]);
   }


      }
       if (state->run_count == 3)
      {


  DEBUG_printf("Writing %d bytes to server\n", state->buffer_len);
```

```
        err_t err = tcp_write(tpcb,state ->  alice_ephemeral_public_key, state-
>buffer_len,TCP_WRITE_FLAG_COPY);
    unsigned long long c_length = 16;
    unsigned char result[c_length];
    for (int i = 0; i < c_length; i++)
    {
        result[i] = state->buffer[i];
    }


    for(int i=0; i< c_length;i++)
    {
      printf("%02X|", result[i]);
    }
    printf("\n");


    unsigned long long clen = sizeof(result);
    unsigned char k[16];


    for (int i = 0; i < sizeof(k); i++)
    {
        k[i] = state->hex_hkdf_output_alice[i];
    }
     //encrypt(sampleResult, M_SIZE, state -> hex_hkdf_output_alice, c, &c_length);
     decrypt(result, clen , k);
      }
    return ERR_OK;
}


return ERR_OK;
```

```
}


static bool tcp_client_open(void *arg) {
    TCP_CLIENT_T *state = (TCP_CLIENT_T*)arg;
    DEBUG_printf("Connecting to %s port %u\n", ip4addr_ntoa(&state->remote_addr),
TCP_PORT);
    state->tcp_pcb = tcp_new_ip_type(IP_GET_TYPE(&state->remote_addr));
    if (!state->tcp_pcb) {
        DEBUG_printf("failed to create pcb\n");
        return false;
    }


    tcp_arg(state->tcp_pcb, state);
    tcp_poll(state->tcp_pcb, tcp_client_poll, POLL_TIME_S * 2);
    tcp_sent(state->tcp_pcb, tcp_client_sent);
    tcp_recv(state->tcp_pcb, tcp_client_recv);
    tcp_err(state->tcp_pcb, tcp_client_err);


    state->buffer_len = 0;


    // cyw43_arch_lwip_begin/end should be used around calls into lwIP to ensure correct
locking.
    // You can omit them if you are in a callback from lwIP. Note that when using
pico_cyw_arch_poll
    // these calls are a no-op and can be omitted, but it is a good practice to use them in
    // case you switch the cyw43_arch type later.
    cyw43_arch_lwip_begin();
    err_t err = tcp_connect(state->tcp_pcb, &state->remote_addr, TCP_PORT,
tcp_client_connected);
    cyw43_arch_lwip_end();
```

```
    return err == ERR_OK;
}

// Perform initialisation
static TCP_CLIENT_T* tcp_client_init(void) {
    TCP_CLIENT_T *state = calloc(1, sizeof(TCP_CLIENT_T));
    if (!state) {
        DEBUG_printf("failed to allocate state\n");
        return NULL;
    }
    ip4addr_aton("192.168.0.119", &state->remote_addr);
    return state;
}

void run_tcp_client_test(void) {
    TCP_CLIENT_T *state = tcp_client_init();
    if (!state) {
        return;
    }
    if (!tcp_client_open(state)) {
        tcp_result(state, -1);
        return;
    }
    while(!state->complete) {
        // the following #ifdef is only here so this same example can be used in multiple
modes;
        // you do not need it in your code
#if PICO_CYW43_ARCH_POLL
```

```
        // if you are using pico_cyw43_arch_poll, then you must poll periodically from your
        // main loop (not from a timer) to check for WiFi driver or lwIP work that needs to be done.
        cyw43_arch_poll();
        sleep_ms(1);
#else
        // if you are not using pico_cyw43_arch_poll, then WiFI driver and lwIP work
        // is done via interrupt in the background. This sleep is just an example of some (blocking)
        // work you might be doing.
        sleep_ms(1000);
#endif
    }
    free(state);

int main() {
    stdio_init_all();
     for(int i = 5; i > 0; i--) {
   sleep_ms(1000);
    printf("%d",i);
    if(i > 1) printf(", \n");
    else printf("\n");

  }

    if (cyw43_arch_init()) {
```

```
        DEBUG_printf("failed to initialise\n");
        return 1;
    }
    cyw43_arch_enable_sta_mode();

    printf("Connecting to WiFi...\n");
    if (cyw43_arch_wifi_connect_timeout_ms("TP-Link_A15C", "43193455",
CYW43_AUTH_WPA2_AES_PSK, 50000)) {
        printf("failed to connect.\n");
        return 1;
    } else {
        printf("Connected.\n");
    }
    run_tcp_client_test();
    cyw43_arch_deinit();
    return 0;
}


void get_shared_key(unsigned char *dh_final, SHAversion whichSha, const unsigned
char *salt, const unsigned char *info,
     unsigned char* output_key, int okm_len){
    int salt_len; //The length of the salt value (a non-secret random value) (ignored if
SALT==NULL)
    int info_len; // The length of optional context and application (ignored if info==NULL)
    int ikm_len; //The length of the input keying material
    uint8_t okm_integer[okm_len]; //output keying material - okm
    ikm_len = 96;
    // printf("%d\n", ikm_len);
    if(salt == NULL) salt_len = 0;
```

```
    if(info == NULL) info_len = 0;
    if(hkdf(whichSha,salt,salt_len,dh_final,ikm_len,info,info_len,okm_integer,okm_len)
== 0)
    {
        printf("HKDF Shared secret):\n");
    } else {
        fprintf(stderr, "\nHKDF is invalid\n");
    }

    for(int i=0; i<okm_len;i++)
    {
        output_key[i] = okm_integer[i];
        // printf("%d\n", output_key[i]);
    }
}
void decrypt(const unsigned char* c, unsigned long long clen, const unsigned char* k){
    unsigned long long m_length = 8;                        // plaintext length
    unsigned char m[m_length];                              // plaintext
    unsigned long long *mlen = &m_length;                   // plaintext length pointer
    const unsigned char ad[] = {0x00};                      // associated data
    unsigned long long adlen = sizeof(ad);                  // associated data length
    unsigned char *nsec;                                    // secret message number
    const unsigned char npub[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0A, 0x0B};  // public message number

    crypto_aead_decrypt(m, mlen, nsec, c, clen, ad, adlen, npub, k);

    // print
    printf("Plaintext = ");
```

```
    for (int i = 0; i < m_length; i++){
        printf("%c", m[i]);
        //printf("%02X", m[i]);
    }
    printf("\n");
}
void encrypt(const unsigned char* m, unsigned long long mlen, const unsigned char* k){
    unsigned long long c_length = 80;                       // ciphertext length
    unsigned char c[c_length];                              // ciphertext
    unsigned long long *clen = &c_length;                   // ciphertext length pointer
    const unsigned char ad[] = {0x00};                      // associated data
    unsigned long long adlen = sizeof(ad);                  // associated data length
    const unsigned char *nsec;                              // secret message number
    const unsigned char npub[] = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
0x09, 0x0A, 0x0B};  // public message number


    crypto_aead_encrypt(c, clen, m, mlen, ad, adlen, nsec, npub, k);


    printf("\n");
    printf("Ciphertext = ");
    for (int i = 0; i < c_length; i++){
        printf("%02X", c[i]);
    }
    printf("\n");
}
```

APPENDIX C

```
// open source glucose meter code, version 1

int strip_detect = 2;
int current = 0;
// measurement starts if transimpedance amp output voltage > threshold
float threshold = 2.8;

#include <Wire.h>
#include "RTClib.h"
RTC_Millis RTC;

void setup() {
  Serial.begin(9600);
  // set the RTC to the date & time this sketch was compiled
  RTC.begin(DateTime(__DATE__, __TIME__));
  pinMode(strip_detect, INPUT);
}

void loop() {
  while(1) {
    if(digitalRead(strip_detect) == 1) break;
  }
  Serial.println("APPLY BLOOD");
  float current_voltage;
  while(1) {
    current_voltage = analogRead(0) * (5.0 / 1023.0);
    if(current_voltage > threshold) break;
```

```
  }
  // count down timer
  for(int i = 5; i > 0; i--) {
    delay(1000);
    Serial.print(i);
    if(i > 1) Serial.print(", ");
    else Serial.println("");
  }
  // compute concentration
  current_voltage = analogRead(0) * (5.0 / 1023.0);
  float concentration = 495.6 * current_voltage - 1275.5;
  Serial.print(concentration);
  Serial.println(" mg/dL");
  display_time();
  while(1) {
    if(digitalRead(strip_detect) == 0) break;
  }
  delay(1000); //debounce
}

void display_time() {
  DateTime now = RTC.now();
  Serial.print(now.year(), DEC);
  Serial.print('-');
  Serial.print(now.month(), DEC);
  Serial.print('-');
  Serial.print(now.day(), DEC);
  Serial.print(' ');
```

```
Serial.print(now.hour(), DEC);
Serial.print(':');
Serial.print(now.minute(), DEC);
Serial.print(':');
Serial.print(now.second(), DEC);
Serial.println(); [10]
```