Secure communication channels for the mTask system

Bachelor's Thesis

Michel de Boer

June 2020

Supervisor:
dr. P.W.M. Koopman

Daily Supervisor:
M. Lubbers

Second reader:
prof. dr. P. Schwabe

Radboud University

Abstract

IoT devices have limited resources available and if we want to change code later on we almost always need to update the firmware. One of the alternatives is using mTask to dynamically send tasks from a central server to the devices to interpret. mTask is currently using plaintext for communication, in this thesis we propose a more secure communication protocol for resource restricted devices. We use the mTask system specifically as an example of such a restricted environment. Using Elliptic Curve Cryptography for signatures, SHA256 for hashing, and ChaCha20 as symmetric encryption, we managed to provide integrity, confidentiality and authentication. Tasks sent from a server to an IoT device can be verified and decrypted within a second. On slower devices such as the Arduino Mega, it takes only around 5 seconds, which can be improved much further. The limiting factor is the Elliptic Curve Cryptography used, contributing to almost all computational resources used.

Acknowledgements

This research would not have been possible without all the help. I wish to express my sincere appreciation to my first supervisor, Pieter Koopman. He guided me to the finish line, providing plenty of feedback along the way. I wish to thank my daily supervisor, Mart Lubbers, for writing most of the server side implementation of the research, as well as helping with technical questions and for providing plenty of feedback to not only the thesis, but also the programming aspects. In addition, he also helped me with all the LaTeX issues I had.

I would also like to acknowledge the help from Peter Achten. For managing the thesis course, but also for providing us with valuable lectures about writing a thesis and keeping up with the schedule. Furthermore, I want to show my appreciation to the second reader, for providing valuable feedback.

Additionally, I would like to thank my classmates, for asking good questions during the presentations, as well as providing different perspectives for my research, by showing their own research.

Lastly, I thank my friends and family for supporting me through the stressful time of writing a thesis.

Contents

Chapter 1

Introduction

1.1 Problem statement

Internet of Things (IoT) devices are becoming more and more popular. With the use of IoT devices comes a certain risk, as these device become a potential attack vector. The IoT devices have limited memory and computational resources, meaning security is often a second thought. If we wish to add new functions to the device or change functions, we may need to send new firmware. For some devices this means we need physical access to a device, as well as needing quite some time to update the device. Another solution is the mTask system. The mTask system dynamically sends tasks to the IoT devices. The devices interpret these tasks. These devices, such as Arduino microcontrollers, now can store the tasks temporarily in memory, waiting for new tasks from a central server. This is faster than needing to do a manual firmware update each time. Lubbers [16] described how to program all layers of IoT using mTask, a TOP implementation for microcontrollers. Part of this is done by sending tasks as bytecode from a faster server to the more restricted devices. Besides tasks, a device can also have Shared Data Sources (SDS). These SDSs can also be shared with the server. The server can then share this SDS with other devices. The mTask DSL restricts the datatypes and uses strict evaluation. This means that we have a concise control flow, making it interesting from a security standpoint.

Currently, the communication is done via plaintext over unsecured channels, such as over a TCP connection or a serial connection. This means that devices by design can be given arbitrary code to be executed by an attacker, as long as it can be described in the bytecode. Moreover, a malicious attacker could change SDS values send to the server. This is a problem if this means that an attacker can interact with real-world things. Such as opening a door or enable a heater.

Most existing solutions that provide fully secured communications over an unsecure channel require a lot of computational power [19]. In this thesis we research different existing solutions of cryptography. We implement a solution based on security principles given in Section 2.3. We use the mTask system specifically as an example of such a restricted environment. Furthermore, the general performance of the system is tested.

1.2 Structure of the thesis

In Chapter 2 it is described how mTask works, as well as how the communication is handled in the original mTask system. Furthermore, some basic concepts will be described. Chapter 3 lists the requirements that we had to keep in mind during the research. This chapter also provides the foundation for the design choices made.

Chapter 4 first motivates design choices and explains how we use the algorithms. Afterwards it gives an overview of the scheme and shows how we tested it's performance, as well as give the results. Chapter 5 lists and describes the related work. Lastly, in Chapter 6 we will conclude the research with a conclusion and a discussion for future work.

Chapter 2

Preliminaries

In this chapter we give the reader the necessary context. We first give some definitions of terms used. We then provide intuition of the system that this thesis builds upon. Furthermore, we describe some basic cryptography principles used. There is a small section about hardware security, which we mostly omitted during the rest of this paper. At last, the communication channels, which we wish to secure, are described only briefly.

2.1 Definitions

Some definitions are needed first. Most of the Task definitions are taken from a paper by Lubbers [16].

- Device, Client: These terms are used interchangeably and denote the actual microcontroller connected to the system. This can be a microcontroller, but it can also just be a program on the same machine as the server, functioning as a client.
- Shared Data Sources (SDS): Data can be shared between the server and devices. SDS is a solution that allows this [16].
- Server, iTasks system: This is the actual executable serving the iTasks application. The system contains tasks taking care of the communication with the clients and infrastructure to manage the clients.
- System, mTask system: The system describes the complete ecosystem, containing both the server and the clients including the communication between them. Depending on the context we refer to either the old mTask system or the modified system, with the security.
- Engine: The runtime system of the client is called the engine. This program handles communicating with the server and runs the interpreter for the bytecode representing the tasks on the client
- Lightweight Cryptography: Cryptography specially designed to be used for computational and/or memory constrained computers.

- Rogue client. A client that an attacker has full control over. It can manipulate messages, change control flow of the client and do everything with the client. This could be for example a compromised client.

2.2 TOP, mTask and iTask

Task Oriented Programming is a new programming paradigm, described by Achten, Koopman, and Plasmeijer [1]. The abstract of the paper describes what exactly TOP is. In summary the abstract described the following:

> *"Task Oriented Programming (or TOP) is a new programming paradigm. It is used for developing applications where human beings closely collaborate on the internet to accomplish a common goal. The tasks that need to be done to achieve this goal are described on a very high level of abstraction. This means that one does need to worry about the technical realization to make the collaboration possible. The technical realization is generated fully automatically from the abstract description. TOP can therefore be seen as a model driven approach. The tasks described form a model from which the technical realization is generated."*

iTask is the system that implements this TOP approach in the programming language Clean[1]. iTask also offers a web service for the end-users with information about the tasks.

mTask [16] is used to allow specialized IoT tasks integrated in iTasks to be run on IoT devices such as the Arduino Uno and the ESP8266. This is done by compiling the tasks into bytecode. This bytecode is send to a device and interpreted.

2.3 Cryptography principles

In this research we assume the reader to be familiar with the basics of cryptography. We only give a short introduction into different terminology and technology used throughout this paper. This includes random numbers, encryption, digital signatures, hashing algorithms, and hardware shields.

Random numbers

Some cryptography needs true random numbers. This means that the numbers are truly unpredictable. True randomness is sometimes hard to check, as it may be very difficult to see whether something is truly random. Some Pseudo Random Number Generators (PRNG) are used for non-cryptographic use. However, those are deterministic, i.e. predictable, and thus not suitable for all cryptography. Some real-life entropy can be used to generate random numbers, or the initial value of the PRNGS, the so called

[1]https://clean.cs.ru.nl/Clean

seeds. True random means that one cannot determine the next random number, even if we know all previous random numbers. Furthermore, if we wish a fair distribution of random numbers, each next bit must have a 50% chance to be a 0 and 50% to be a 1. This can sometimes be difficult, as not all noise is distributed this way. Consequently, sometimes post processing is needed. Even when it is difficult to check if something is truly random, we still rather use a cryptographically secure pseudo random number generator, because cryptography often require secure random number generators instead of PRNGs.

Encryption and decryption

Encryption is the process of turning a readable plaintext into a ciphertext, in such a way that an attacker without the key cannot know what the original plaintext is. Decryption is the process of transforming the cipher back into the plaintext. One can only decrypt correctly if the key is known. There are two main flavours of encryption, namely asymmetric and symmetric.

Asymmetric and symmetric cryptography

Symmetric encryption uses the same key for both encryption and decryption. This key must be kept secret for all other parties not participating. Asymmetric cryptography, also called public key encryption (PKE), uses two different keys, one for encryption and one for decryption. Most of the time we have one private and one public key. The public key can be given to anyone. Depending on the algorithms used, Alice can for example use the public key of Bob to send a message to Bob. Only Bob has the private key, meaning only he can decrypt the message. Most of the times both Alice and Bob have their own private key. This way we get a two way channel, where Alice can encrypt with Bob's public key and Bob can encrypt with Alice private key.

Digital signatures

Digital signatures can be used to verify that a message is indeed from a certain person. They can be generated with a private key. For example, Bob can sign a hash of his email with his private key. Alice can now verify with Bob's public key if someone else tampered with the email. Furthermore, digital signatures can also provide non-repudiation. If only Bob has the private key, and we know that the message is signed with the privater key, then Bob must have send the email.

For both digital signatures as well as public key cryptography. It is essential to make sure that the public key is indeed from the person you think it is. In a webbrowser this is done by a chain of trust. There are several public keys stored by default in your browser (or operating system). The corresponding private keys can be used to sign other public keys, thus creating a chain of trust. This only works because the public keys stored on your computer are assumed to be trusted. Another way to verify each others public key is to exchange a hash of the public key over a secure channel, for example in person on a paper.

Hash functions

Hash functions take a variable-length input message and irreversibly turn it into a fixed-size output, called a hash or digest. Cryptographic hash algorithms are secure versions of hash algorithms. For example, changing one bit in the input should change each bit in the output with a 50% probability. Furthermore, having different inputs results in different outputs with very high probability. A hash algorithm is said to be collision resistant if it is difficult to find two different inputs, that result in the same hash.

Since hash algorithms have a fixed output size and a variable length input, we can not always prevent a collision, because of the pigeon hole principle. Generally, a hash algorithm has a certain security against collision attacks, expressed in bits.

MACs

A Message Authentication Code (MAC) is a fixed size tag appended to a message. It is used to check the integrity of a message. The same key is used to create the tag as well as verify the tag. Without the key, one should not be able to do either of them.

Additionally, some MACs use a hash algorithm turn a cryptographic hash function and a key into a MAC. There are different ways to do this. We describe three possible MAC's. The first one is susceptible to length-extension attacks. We will also describe this attack. The second MAC scheme has issues if the hash algorithm turns out to be broken. The last one is the widely used and accepted HMAC.

The first scheme is to prepend the key to the message. The issue with this is that some underlying hash algorithms suffer from length-extension attacks. Examples of hash algorithms that suffer from length-extension attacks are SHA256, MD5 and most other Merkle–Damgård based hash algorithms. When given a hash, one can construct its internal state at the end of the hash algorithm, because the internal state is exactly what the output of the hash algorithm is. In the next section we describe the attacks that such Merkle–Damgård based hash algorithms are susceptible to.

Length-extension attacks

We will first describe how length-extension attacks are an issue for hash algorithms. Let's say we simply prepend the key to the message and use this as our MAC.

$$HM1 := Hash(key\|message)$$

The length-extension attack now allows us to compute a new hash as follows, without knowing the key. We first set $HM1$ as the internal state of the hash algorithm. An attacker can then calculate the hash over a forged message.

$$HM2 := Hash_{HM1}(rogueMessage)$$

Now, this is exactly the same as if a valid MAC would be calculated, but we have to keep into mind the padding used.

$$HM2 = Hash(key||message||padding||rogueMessage)$$

In our case, this means an attacker can append additional bytecode to the message. This is solved altogether by using digital signatures instead of just a MAC in the mTask system. In the next section we describe another solution, which can be used if digital signatures are not an option.

MAC schemes

In the previous section we described an issue with prepending a key. Instead of prepending the key to the message, we can append it to the message, as follows:

$$Hash(m1||key)$$

However, there is an attack possible if the hash algorithm is not very collision resistant. That is, if we can find:

$$Hash(m1) = Hash(m2)$$

Then we also found:

$$Hash(m1||key) = Hash(m2||key)$$

There are many more variations of MAC. One widely used is HMAC, described in RFC2104 [21]. Updates to this Request for Comments describe mostly security considerations for the MD5 hash algorithm. Section 2 of RFC 2104 states:

```
"We define two fixed and different strings ipad and opad
 as follows:
   (the 'i' and 'o' are mnemonics for inner and outer):
                  ipad = the byte 0x36 repeated B times
                  opad = the byte 0x5C repeated B times.
   To compute HMAC over the data 'text' we perform
                    H(K XOR opad, H(K XOR ipad, text))"
```

As we can see here, we hash twice, with inner and outer padding in between.

2.4 Hardware security extensions boards

Kristinsson et al. concluded that Arduino Duemilanove boards themselves do not provide enough entropy to be used as a random number device [14]. They show by statistical methods that the atmospheric noise of the Arduino is mostly predictable in certain situations. They also explore different methods to extract true randomness. They conclude that the analog pins should not be used as a source of randomness.

One of the solution for this is to use a hardware extension board. However, in this research we wish to use as little external hardware as possible, as it can be complex and limits the available pins for peripherals. Furthermore, we decided to not make use of any algorithms that need random numbers on the devices. Nonetheless, we wish to present the reader a general idea of what is possible.

There are hardware extension boards for some devices that can generate random numbers. These shields can rely for example on thermal noise from a resistor or atmospheric noise such as electromagnetic radiation. There are many more sources of randomness, such as clock drift or even some quantum properties such as radioactive decay. Using shields fixes the randomness on our device mostly. However, one has to take into account that an attack that can access the shield, can potentially modify the random stream. For example, if the shield sends the results to the device via analog/digital IO pins, an attacker can simply remove the shield and send their own numbers through the pins. It could be that there are shields that solve this issue by preventing physical access, or detecting when someone opens up the device. One of the hardware security shield devices is the CryptoShield[2]. This open source shield can run different software. It can run SHA256 as well as some type of Elliptic Curve Digital Signature Authentication (ECDSA). The microchip used is the ATECC108[3].

2.5 Communication channels

Before this research thesis alters the communication of the mTask system, the communication is done as described in this section. The server manages all devices. Only the server is capable of initiating a connection with a device [16]. Multiple devices can be connected to a server, but a device can only be connected to a single server. Only the server should able to send tasks to a device. Furthermore, SDSs are used mostly for communication.

In Appendix A of a paper by Lubbers more information can be found about the communication [16]. A summary is given below. The communication starts with a handshake. Followed by one or more of the other actions. A general overview is given below. A lot of things changed during the development of the mTask system, below is the version before added encryption.

- **Handshake**: The server sends one byte with value ‘c’ to a device. The device will respond with 8 bytes of information.
- **Send/receive mTask tasks:** The server sends a request of n+6 bytes to the device. Here n denotes the length of the bytecode send. A device responds with 3 bytes.
- **Delete mTask tasks:** The server sends a request of 3 bytes to remove a task. The device responds with 3 bytes.

[2]https://github.com/sparkfun/CryptoShield

[3]http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8873S-CryptoAuth-ATECC108-Datasheet-Summary.pdf

- **Send SDS specifications:** The server sends a request of 5 bytes. This includes the value of the SDS. Device responds with 3 bytes.
- **Delete SDS:** The server sends a request of 3 bytes. The device responds with 3 bytes.
- **Update SDS:** The server sends a message of 5 bytes. This includes the value of the SDS.
- **Publish SDS:** The device sends 5 bytes to the server. This includes the value of the SDS that the device changed. Note, in the current version one can send larger SDS to the server. In that case the number of bytes is variable.

Chapter 3

Security principles

To be able to define a secure channel we will first need to discuss the different security principles that play a role. The security principles for IoT are described in more detail by Mahmoud et al [17]. In Section 3.1 we make a selection of the security principles we consider most important for our research.

- **Confidentiality** No other party can read the contents of the messages without a key.
- **Integrity** The message has not been tampered with.
- **Authentication** It is verifiable that a message came from a specific instance.
- **Availability** The system remains available for use.
- **Authorization** It is defined who are allowed certain tasks. This can be done for example via permissions.

3.1 Security assumptions and scope

Unfortunately, it is not possible to research all security aspects at once in this thesis. Therefore a scope is defined of the research. The research is oriented to the security of the communication channels between client and server. This means that aspects such as the physical security of devices will not be taken into account. Additionally, it is assumed that the server is reasonably secure. That is, in such a way that key management is taken care of as well as well as the integrity of the control flow. It is however taken into account that the devices could be less secure overall. We assume that an attacker could at least read everything from a device. This attack model is chosen because hardware security is not yet very well considered in this research paper.

3.2 Security requirements

Not all security principles are implemented. In this section our most important requirements are described.

- **Integrity:** The client must be able to verify the integrity of tasks and SDSs send from the server. That is, it should be clear when a message has been tampered with.
- **Authentication:** A rogue client should not be able to impersonate honest clients or impersonate the server. A possible solution is to let the server and client authenticate each other.
- **Lightweight:** The clients should not have to do lots of computations or use lots of memory for the cryptography used.
- **Other: MITM** A man in the middle attack (MITM) should not be possible on the communication channel.
- **Other: Initialization vectors (IV).** IVs are used to initialize encryption modes or algorithms. Depending on the algorithm these needs to be truly random. Therefore these should not be generated on devices. Furthermore, IV should not be reused with the same key. For each message another IV should be used, if an IV is needed, as most encryption schemes require this uniqueness.

There are some more requirements, which are not directly required for iTask/mTask, but are good to have in other use cases.

- **Authorization** Some devices may not receive or update all SDSs. Some sort of access control on the server would be a good addition.
- **Confidentiality** Confidentiality should be possible. However, making this optional per message could be a compromise between security and computational resource, depending on the requirements.

The availability security aspect is chosen to be ignored. Mostly because the resources are very limited on the clients, making it a difficult task, requiring research on its own. Furthermore, we leave authorization as a future research.

3.3 Device limitations and design criteria for the system

We need to use some primitives to build our protocol. Out of the existing lightweight cryptography primitives, we make a selection based on some criteria and limitations. The first criterion is resources. When a primitive already uses too much resources, a protocol based on it will never be fast enough, or not fit in memory. To decide what resources are available we first give the important specifications of the different devices. After giving those resource specifications we will introduce further limitations, as well as the main design criteria that were used during the development of the system.

Table 3.1: Maximum memory capacity of different devices.

	Mega2560	Uno	nano328	ESP8266
flash	256KB	32KB	32KB	512KB/1MB/4MB
sram	8KB	2KB	2KB	80KiB + 16KiB Wi-Fi
eeprom	4KB	1KB	1KB	4KB

Device resource specifications

An Arduino Uno uses the ATmega328 processor. It runs at only 16 MHz[1]. The processor of the ESP8266 operates at 80 MHz[2]. Also keep in mind that the Arduino Uno operates on an 8-bit architecture, while the ESP8266 has a 32-bit architecture.

The Arduino nano328 is similar to the Uno, but is physically a bit smaller. The Arduino Mega2650 on the other hand is much bigger. It also has many more I/O pins as well as much more memory. Table 3.1 shows the maximum memory for the devices.

3.3.1 Limitations

Besides fitting in the Arduino Uno memory, another limitation is the use of random numbers.

Due to low entropy, generating random numbers for the Arduino seemed to be a problem in previous research [14]. Therefore it is better to let the server handle the generation of random numbers. Another option is to design the protocol in such a way that the devices do not need random numbers. We made sure to select only algorithms that do not need random numbers on the device side.

3.3.2 Design criteria

An important principle was to keep the protocol as close to the existing protocol as possible. That is, we do not want to introduce many new messages in the protocol. Doing so hampers backwards compatibility. Furthermore, modularity is also very important. We wish to be able to swap out algorithms very easily later on.

[1]https://www.microchip.com/wwwproducts/en/ATmega328
[2]https://docs.zerynth.com/latest/official/board.zerynth.nodemcu2/docs/index.html

Chapter 4

Design

In this chapter we dive into the technical details of the system. In section 4.1 we will first introduce the algorithms we use, as well as give the reasons we decided to use them. Furthermore, we describe how we worked towards a scheme as well. In section 4.2 we give an overview of the protocol used for the mTask system. Lastly, in section 4.3 we describe how we decided the parameters, as well as how we tested the performance of the system. The results of the tests are described in section 4.4.

4.1 Design choices

In summary, we choose ChaCha20 as it is a stream cipher that uses very little memory and is fast. ChaCha20 requires nonces instead of random IVs. Which means we will not need random numbers on the devices. For digital signatures we use Elliptic Curve Cryptography, with the curve secp256r1. For hashing the message before signing, we use SHA256.

The next sections describe in more detail how and why we choose those algorithms. Additionally, we also explain how we manage the nonce that is required by ChaCha20. In the section after that, we describe how we provide limited forward secrecy.

4.1.1 Encryption

Before we can decide exactly what encryption algorithm to use, we must see what the possibilities are. For encryption we have two main categories. The first category is symmetrical key cryptography and the second is public key cryptography. Within symmetrical key cryptography we have two subcategories, they are block ciphers and stream ciphers. We will describe them each individually and motivate why we choose to use a stream cipher. In the section about public key cryptography we describe why this was not an option for the mTask system.

Block ciphers

Block ciphers work on encrypting data in blocks. AES for example works on blocks of 16 bytes. Thus this is the minimum size of a message that can be send.

There are several block ciphers that can be used. Cryptolux lists Chaskey[18], Speck[4] and AES[7] as recommended block ciphers[6]. The first two are in almost all scenarios rated high[6]. AES is rated good in execution time and RAM used.

Stream ciphers

Stream ciphers can take messages of any length. Stream ciphers do however need to be implemented carefully, as the plaintext is XOR'ed with the stream. I.e. if we have a nonce reuse we could leak at least partially the data with a stream cipher.

Only three lightweight stream ciphers seem to be tested by Cryptolux [6]. Trivium [8], ChaCha20 and Snow3G. About Snow3G not a lot of information can be found and it is rated below the other two stream ciphers on CryptoLux [6].

Public key cryptography (PKC)

In general, public key cryptography is much slower than symmetric cryptography (See Chapter 5). As a result of the heavy computations needed, PKC is not suitable for the use of encrypting every message in the communication channels.

ChaCha20 as choice of encryption

When using a block cipher or stream cipher, it is crucial to handle IVs or nonces. For example, the output feedback mode (OFB) and counter (CTR) mode for block ciphers need unique and random IVs. This is fine if the server sends the IV. However, the device cannot generate random IV's. When the server initiates a message sequence, we can simply send a new IV with it. However, when a device first sends the message it does not yet have an IV. A full nonce collapse is a bigger problem with stream ciphers, because it uses XOR'ing we may leak partial plain text.

This includes OFB and CTR, as those modes make a stream cipher out of a block cipher. Although nonce reuse can be a problem for stream ciphers, they do not need padding. CBC mode requires padding, but does not have a full collapse if we have a nonce reuse. CBC requires an IV mostly for the "chosen plaintext attack". CTR mode does however only need nonces as IV. This means that we do not have to randomly generate the IV, as long as we make sure to never re-use the IV. Thus making CTR mode very suitable for our need. Furthermore, we can combine CTR mode with CBC-MAC. This combined mode is called CCM mode.

When using CCM mode, we can reuse the same key for both MAC and encryption as long as the IV used for the MAC is not the same as the nonce for the encryption. Note that it is possible to use a static IV for CBC-MAC. Using a static IV will however remove the ability to prevent replay attacks. CBC-MAC is not secure on variable length messages. This can be overcome by specifying the length of the message in the first block, as is done in CCM.

We decided to use ChaCha20 as our choice for encryption. The first decision was to use a stream cipher instead of a block cipher. This is done because it makes it very easy to use per byte, rather then per block. This is very useful as the original code was oriented towards reading per byte. Although ChaCha20 internally uses a 64-byte size block, it is still relatively lightweight in its implementation. The main reason to use a stream cipher over a block cipher is that we will not have to take care of mode of operation. In ChaCha20 this is implicitly done for us, by using an internal counter. Even though block ciphers are similar in performance, we would have to implement a mode of operation ourselves or use an existing one. Using a block mode means that we have to use more code. Some block modes cannot be used as they require random numbers.

Managing the nonce

ChaCha20 requires a 128 or 256-bit key, as well as an 8-byte (or 12-byte) Initialization Vector (IV), which is a nonce.

The IV, called nonce in this case, must be unique for each message, but must not be random. The reason for this is that the 64-bit IV is too short to be a random number. This is explained using the birthday paradox. Meaning approximately 2^{32} messages are needed for a collision in a random 8-byte (64-bit) nonce[1].

Each device keeps a counter of the nonce. When receiving or sending a message it increments the counter appropriately. The server does this as well, having a nonce counter for each device. The communication channels are reliable, it is however possible that a device and the server get out of sync for the nonce counter. For example, because of a power shortage on the device during decryption. This can be fixed by doing a key schedule. More information about key schedules can be found in a later section, but for now it suffices to know that a key schedule resets the nonces we can use. This is because a key schedule makes sure that the device uses a new key. Nonces only need to be unique with the given key that is used. We do need to make sure that we will not re-use a nonce for this key schedule however. This can be done easily by starting each nonce at 2 after a key schedule. This way we reserve the nonce 0 and 1 for a key schedule, thus we can always do a key reschedule.

All other numbers within the range of the nonce may be used for any other messages. This means that the sequence of nonces are as follows: $(A_n)_{n=0}^{end} = \{2, 3, 4...., (END - \mathit{2}), \mathit{0}, \mathit{1}\}$. Where *END* denotes the maximum number of messages send before we run out ot nonces. This depends on the size of the nonce. We can reset the nonces we use by doing a key schedule, as described later on.

As a result of the design, a key schedule may only be done once per key. This is indeed the case, as seen in section 4.2.2. Each message needs 2 nonces, if in later versions we need more messages reserved for the key scheduling, we need more nonces. We can

[1]RFC 7539: https://tools.ietf.org/html/rfc7539

simply change the starting value for the nonce to some greater value in that case.

Having only one key schedule per key is not an issue, as we will have a new key afterwards, which resets the nonces we can use.

Not only ChaCha20 needs a nonce. The digital signature should also have a nonce to guarantee freshness. Although the digital signature and the encryption c.q. decryption are very different with how they use the nonce, we decided not to use reuse the nonce in one or the other. That is, a number may be used exclusively for the digital signature or for the encryption. This way, we do not rely on the internal structures of the hash and encryption. It may happen to be that the internal structure is such that a nonce reuse between the two exposes some information. As far as we are aware, this is not the case with the algorithms we use. However, if we wish to change the algorithms used, we may not always be able to assume this.

By design we want to first check the authentication then only decrypt the message. Thus the counter for the decryption is always one higher.

All of this comes at the cost of needing to have a key schedule sooner. When exactly we do the key schedule can be easily configured. It is a trade-off between performance and how much forward secrecy we provide.

Confusingly, ChaCha20 also uses an internal counter besides the nonce described earlier. This is for example to allow parallel decryption of different blocks, or if we got really big messages. We do not need to take this into account, as the number of bytes per message will not exceed 2^{64} (the size of the internal counter), especially in our restricted environment this is a very safe assumption. Therefore we only look at the (nonce, key) pairs.

4.1.2 Forward secrecy

Forward secrecy is a good property to have, as it makes sure that an attacker cannot read back all previous messages if a device is compromised. In this section we will describe how we achieve some sort of forward secrecy. It does not provide full forward secrecy per message, rather until the next key schedule.

With a 64 bit nonce we have plenty of messages that we can send without a key schedule. However, once an attacker gets access to a device, it can read all data back of previous messages. In other words, it does not give forward secrecy. Instead, we may use a smaller nonce and do a key schedule much more often. This way an attacker can only read the data that is encrypted using the current key. However, we must ensure that the previous key is destroyed.

Finding a good value for when to do the key schedule is a trade off between security and performance. Doing a key schedule gives an overhead, as the device has to verify the message as well.

We decided to use a counter of only 16 bits. This means that at first glance we have to do a reschedule at nonce $2^{16} - 2$. However, a signed messages needs 2 nonces. Thus we have to do a schedule after $2^{15} - 1$. The exact value we do this on can be changed any time. However, $2^{15} - 1 = 32767$ messages is still a lot of messages, so we may want to reschedule a key much earlier. Also, we still have 48 bits of the nonce left that we do

Table 4.1: Internal state ChaCha20, using a smaller nonce.

Row	Bytes 1,2,3,4	Bytes 5,6,7,8	Bytes 9,10,11,12	Bytes 13,14	15,16
1	”expa”	”nd 3”	”2-by”	”te”	” k”
2	KEY (2 rows of 16 byte each)				
3					
4	POS (internal ctr)		{Version} 6 *0x00		Nonce

not need. We decided to use this to show the version of our security module, starting with only 0 bytes in the first version (the one that is described in this thesis). Having 0 values in a nonce is not an issue. From a cryptography point of view, the full nonce is 64 bits, whether we use it as 0 or not. As long as we do not reuse the full 64 bit nonce with the given key it does not lead to nonce reuse. One important note must be made. Having some way of guaranteeing which previous key was used during the key schedule is important to have forward secrecy. There are several attacks that could happen otherwise.

For instance, if we did not include the key in the hash, e.g. not using a version of MAC, one could replay a previous key schedule. This is still a valid signature, decrypting with the new key probably fails though. It could be the case that the bytes result in correct bytecode. However, this does mean that every instruction must be valid and pass all the strict evaluations the mTask offers. However, in other IoT systems this may not always be the case. Thus we had to make sure that this does not happen. Similarly, replaying a previous message with the same nonce, but another key, could also lead into issues if we did not use a key in the hash. This is because the signature uses the same nonce, and thus be verified correctly. Once we decrypt, we use another key than we used during encryption in this attack. Thus the resulting plaintext on the device is different from what was originally intended. Including the key in the hash should counter this, as we have different signatures for each key, nonce combination.

The design choice for using smaller nonces results in the internal state of ChaCha20 looking a bit different from normal. The internal state is used to calculate the final cipher, by applying functions to the state and plain text. From a cryptographic point of view, the internal state structure did not change at all. However, the way we use the nonce makes the interpretation a bit different. As the nonce is divided further up into parts. 4.1 shows the new internal state structure, in which row four, bytes in column 13-16, differ from the original state structure.

4.1.3 Hash algorithms used

Choosing the hash function was simple. There are only a few common hash functions that are widely used. Such as MD5, SHA0, SHA1, SHA2 and SHA3 (subset of Keccak). MD5, SHA0 and SHA1 are considered broken. SHA3 is a good contender of SHA2, but uses a larger internal block state and as a consequence it uses a bit more memory, making it a bit less usable for our lightweight solution. Should it happen that SHA2 also gets broken in practice, one can easily change the design to implement any other hash function.

It is important to consider that SHA2 suffers from length-extension attacks. This is not an issue for us, as we sign only the given hash on the server, and compute the hash on the device.

For the design it was also important that it works on bytes rather than blocks, to keep the bandwidth as low as possible. Since there are many small packets send over the channels, adding padding would increment the bandwidth used rapidly. Luckily, most hash functions can be updated per arbitrary number of bytes and therefore also per one byte at a time. The signature can simply be read after the message.

However, as the metadata is read before the original message, an attacker could change the metadata, causing the device to read more data into memory than it should. With metadata, we mean: taskid, returnwidth, sds length, number of peripherals and bytecode length.

Only after reading the data, the device detects that the data was invalid. A simple solution is to first sign the metadata and then sign the actual data This however means that we double the time it takes to verify the message. Another solution is to hard limit the size of a task. This is already done implicitly, as the lengths are specified in 16 bit integers. However, this could still be too much for the device to handle in reasonable time. As not only too much memory has to be reserved, the hash needs to be calculated over it as well. In the case we have a message that exceeds the reserved memory size, the device will report an error back to the server, indicating that the task does not fit in memory.

Another solution is to add a time limit. If not all the data is collected within a time, it says that the verification fails. This however could lead to synchronization problems. Since this research does not focus on availability of the devices, we decided to use the implicit limitation of 16 bit integers.

We decided to first read in the metadata non encrypted, as those are apparent from length of the cipher text anyway. However, the metadata is still authenticated! We then request memory for the task and read the encrypted payload from the communication channel. We verify the task over the metadata and the cipher text. If we succeed, we decrypt the task data per byte. Otherwise we throw the task away. This method is chosen because we need the memory anyway for legitimate tasks and availability is not the most important requirement in this thesis.

4.1.4 Authentication

One of the main requirements to allow authentication is that we have some secret stored on the server and device. It is assumed that arbitrary memory can be read from a device. Either directly, via debug pins on the device or via side channel attacks. Writing arbitrary data to the device is assumed to be much harder for an attacker. Because of these assumptions, a Message Authentication Code (MAC), which uses symmetrical key cryptography, will not work. An attacker could simply read out the key from the device.

Since this will be the same key as on the server, an attacker can pretend to be the server. Thus sending rogue tasks to a device.

This can easily be overcome by using digital signatures. The devices uses a public key, which can be read by anyone. The private key should however be stored securely on the server. This way an attacker cannot modify tasks, as only the server can sign tasks and everyone with a public key can verify the task. We opted to use this authentication method. The device will have the public key, but in principle everyone can have the public key of the server.

There are two commonly used digital signature algorithms that are widely used. RSA and Elliptic Curve Cryptography (ECC).

RSA is generally faster with verification than ECC [12]. Nonetheless, the performance in other areas, such as creating keys and signing is slower [22]. The performance on AVR platforms is not that clear and there are currently no easy to use and good RSA implementations for AVR devices. Furthermore, RSA uses much more SRAM than ECC, meaning a device will run out of SRAM very quickly. Rather than writing difficult implementations ourselves, we use libraries found. We decided to use MicroECC as library for ECC[2]. We use the curve secp256r1 with the given library. This can easily be changed to other curves with compile settings for the library used.

Instead of doing all the computations over the full raw message, we only do it in the end over the application data. This makes it possible to use it for different communication methods. Such as TCP, Serial connection and Bluetooth. This makes sense, as we should abstract away from the underlying transport layer. We only encrypt parts of the application layer. Furthermore, using a hash at the end rather than using the full message for signing and verify is common practise. Signing and verification is very costly, thus doing it over a smaller portion of the message is also preferred.

Message Authentication Code.

Ensuring that a message has not been tampered with can be done using a MAC. A receiver can check if the MAC corresponds to the message received. This gives a message integrity. There is however a special security requirement that makes using a MAC more difficult. We must have some sort of authentication on messages send from the server. Even if we recover the keys from a device, it must not be possible to send a task to that device. This comes from the attack model assumption we use. The assumption is that it is easy to read out any data from the device. However, writing to a device is assumed to be much harder. A MAC provides integrity, but it does not provide any authentication. Therefor it is not something we can use on a device to check if the server indeed sent the tasks. In future versions it may be possible to add a MAC to less important messages. This way we at least guarantee integrity for a man in the middle without access to the device.

We decided to use Micro-ECC as library for the signing and verification of messages, because this library uses faster curves, making it suitable for restricted devices. When

[2]https://github.com/kmackay/micro-ecc

choosing a curve we keep in mind that NIST does not recommend a key size of less then 224-bit for ECC [9].

For the symmetric cryptography and hash functions we used Arduinolibs[3]. Unfortunately, this library is written in C++, meaning it will not work on all devices. Some devices do not work well with C++ compilers. Since we only need one state for SHA256 and ChaCha20, this was easily converted into a C implementation by only changing it from a class to global functions and states. There are lots of different implementations of SHA256 and ChaCha20 for the devices, we choose to use the given library because it implements many different algorithms, making it easy to switch algorithms during development in case we wish to see differences. A drawback is that it is not as optimized. Luckily we abstract away from any third party libraries used. One can simply change the library, only changing the wrapper functions in *Security.h* and *Security.c* to call the new library instead.

4.2 Scheme

In this section we describe how we use the different algorithms to build a scheme. We first describe how we initiate connections and handle initiation. We then describe how we do a key (re)schedule. Lastly, we describe our main interest, sending tasks to a device.

4.2.1 Sending specifications

Only the server can initiate a connection. The servers starts the connection by sending a message to the device. The device responds with what its supported features are. For the peripherals it responds with one byte to indicate if a certain peripheral is supported. We keep the original message flow similar to the original mTask system.

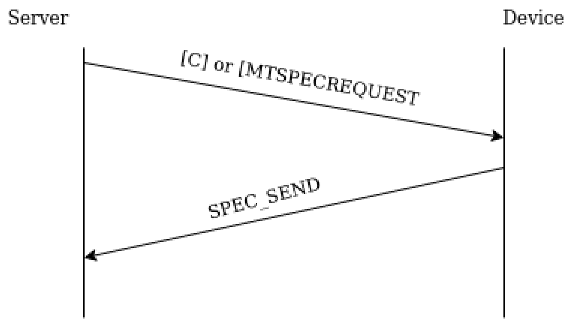

Figure 4.1: Initial handshake.

The `SPEC_SEND` message is changed at first. The first 5 bytes stay the same as earlier: `MTFSPEC`, `MEMSIZE` (2 bytes), `APINS` and `DPINS`. After these bytes, normally a device sends

[3]https://github.com/rweather/arduinolibs

which peripherals are supported. Per peripheral, one byte is sent, having value 1 if the given peripheral is supported by the device and a 0 otherwise, the other 7 bits are not used in the previous version. To keep backwards compatibility, we cannot send more bytes, as the server would not know how many bytes to receive from the device. Instead, if the device supports security it will modify the first byte of the first peripheral. It will send 0b11 if the first peripheral is supported and 0b10 otherwise. The extra bit is thus used to indicate that the device supports the new security extension. Once the server gets this response, it knows it has to use the security extension as well.

By choosing this method we can also allow multiple versions of the security module if ever needed. Simply using the other bytes of the peripherals as well, or even using one bit to indicate that the device will send more bytes afterwards.

The original task message looks as follows, where the *get* function is denoted to indicate that it is the actual bytes (with variable length).

- taskid: A unique task id. Used to identify task. For instance removing tasks.
- returnwidth: The length of the return value in bytes. During evaluation, a task may communicate it's value to the server.
- sdslen: The number of bytes the SDSs takes up in the message.
- peripherals: The number of bytes the the peripherals data send in this message.
- bclen: The length of the bytecode that is includes in this message.
- get(sdsbytes): The actual SDS data that is included in the message. This way, devices are immediately up to date with the SDS information.
- get(peripherals): The peripherals data. This is also called hardware. For example, setting up certain sensors, such as a heart rate sensor.
- get(bytecode): The actual bytecode. The device interprets this data.

4.2.2 Key schedule

Once we run out of nonces, we need to do a key schedule. Should it happen that the server and client get out of sync, for example because of power outage of the device, one must do a key schedule as well. Doing a key schedule often also reduces the chance that leaking a key results in all messages losing confidentiality. Note that only the server can initiate a key schedule. The public key of the server on the device can for now only be set at compile time for the device. The first symmetric key that both the server and device uses should also be stored during setting up the device.

When a key schedule is initiated, the server will simply send the following bytes to the device:

```
hash := sha256((nonce=1, symkey) || newkey)
signature := sign{hash}
Message := MTTKEYSCHEDULE || Ek(nonce=0){newkey} || Signature
```

The nonce and symkey are appended before the other parts of the message.

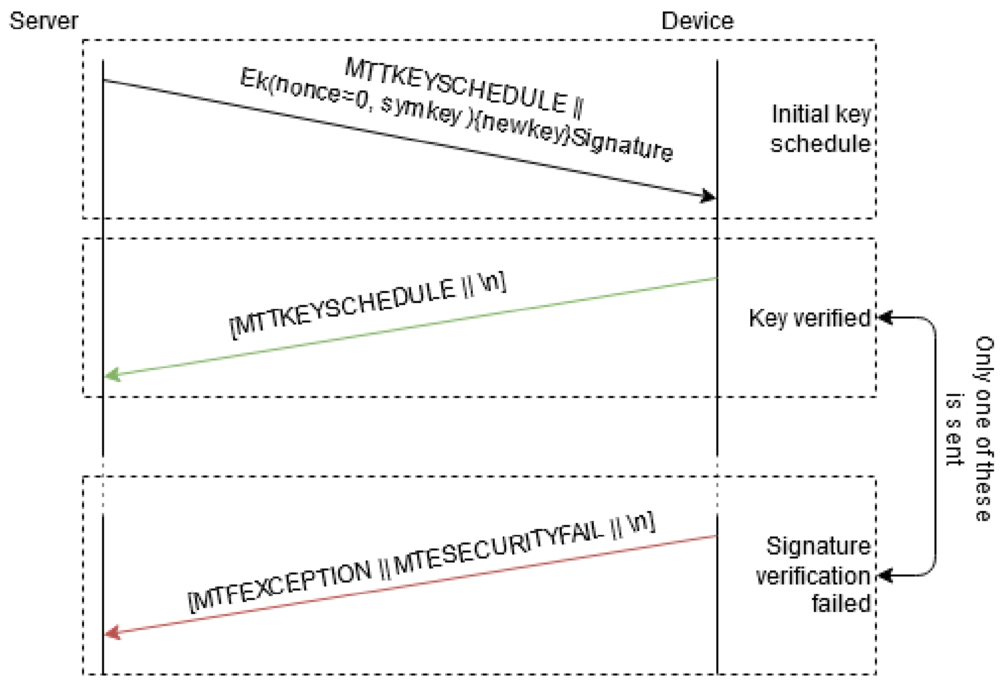
Server
Device
MTTKEYSCHEDULE ||
Ek(nonce=0, symkey)(newkey)Signature
Initial key schedule
[MTTKEYSCHEDULE || \n]
Key verified
Only one of these is sent
[MTFEXCEPTION || MTESECURITYFAIL || \n]
Signature verification failed

Figure 4.2: Key schedule. Server to device.

The device itself is not able to request a new key. This is for several reasons. First of all, securing the request for a new key is difficult, as a device cannot sign messages the same way as the server does. Due to memory and computational limitations, the sign function is not implemented on devices. Using an HMAC is possible, but it is much easier to not allow the device to request new keys in the first place. The only drawback is that a device may run out of nonces when it wants to send a message to the server.

This is currently not an issue, as the client and server communicate often enough in both directions. Allowing plenty of opportunities to do a key reschedule. Additionally, there is another easy solution if we can assume that the communication channel takes care of reliability. Once the server notices that the nonces run out on the device, it can send a key schedule. The device still must make sure that it does not send new messages. The device must wait for a key schedule once it runs out of nonces, the server must notice this based on its own counter. Since we only specify the maximum number of message send before a key schedule, the server could even decide to send a new key much earlier. This assures that the device will not have to wait long.

4.2.3 Sending tasks

Sending tasks has changed a lot more. We need to store tasks in the device memory before we can use them. Tasks must now be verified and decrypted as well. Metadata is not encrypted in tasks, but they are still verified. We first describe how tasks are stored in the memory, then we describe how the different algorithms work together to provide encryption and authentication. Lastly, we describe the encryption in even more detail, as a lot of design decisions had to be made for making sure the requirements of the encryption algorithm are met.

Storing tasks

Since we need to store the task in memory before we can interpret the bytecode anyway, we decided to first read out the task in memory fully. This is done as usual in the mTask system, using `mem_alloc_task`, thus no extra memory is needed to store it. However, at first we load in the encrypted data. We then verify the authentication of the message later on, but before using the message. The metadata is not encrypted, this includes, `taskid`, `sdslen`, `returnwidth`, number of bytes in bytecode and number of peripherals and bytes used for SDSs. The metadata is however included in the verification hash and thus authenticated.

Communication of sending tasks

Figure 4.3 describes the task communication. After the device received the message and signature, it will start to verify this message. In summary, the message consists of non-encrypted metadata, an encrypted message and a signature. The device checks the signature and either accepts or rejects the message.

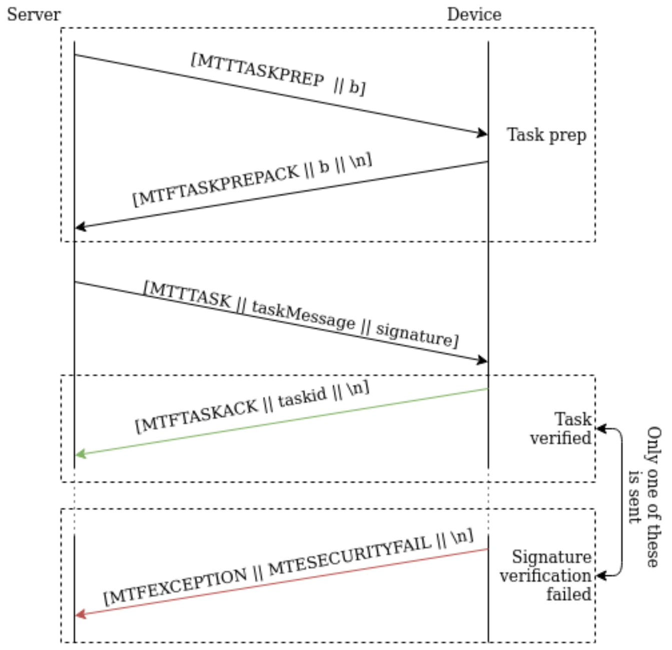
Server
Device
[MTTTASKPREP || b]
Task prep
[MTFTASKPREPACK || b || \n]
[MTTTASK || taskMessage || signature]
[MTFTASKACK || taskid || \n]
Task verified
Only one of these is sent
[MTFEXCEPTION || MTESECURITYFAIL || \n]
Signature verification failed

Figure 4.3: Sending tasks.

```
cipher := Ek(Nonce){get(sdsbytes) || get(peripherals) || get(bc)}
taskMessage := (taskid || returnwidth || sdslen ||
                peripherals || bclen ||  cipher)
hash := sha256({nonce+1, symkey} || taskMessage)
verified := Verify(Pk_server, hash, signature)
```

The notation Ek is used to denote encrypting using key k. In this case, it is a symmetric key, meaning both the device and server share this key. The Pk-server is a public key, meaning everyone may have this key. Only the server has the private key, used for signing the hash. The server follows a similar scheme when encrypting, but instead of verifying the signature it computes the signature and sends it together with the message. The || indicates appending. Note that with a hash algorithm, it does not matter if we append the message before we call the hash function, or if we call hash_update on the first part, then on the second part. That is, the following lines give the same result:

```
// Doing this
hash_update(a.append(b));
// Is the same as doing:
hash_update(a); hash_update(b);
```

4.3 Experimentally decided parameters

4.3.1 Experimentally decided parameters

There are lot's of options to compile the code into the device assembly code. Depending on what settings we use, we will get different performance results. The Arduino Uno has limited memory, so we wish to optimize for that. In contrast, the ESP8266 has much more memory, so we may optimize for speed. To know if the system works on a given device, as well as find the performance bottlenecks, we have to do some experiments. We first test the full system, to see if the entire code fits on the device. We then test the speed of the different algorithms used. Both to verify earlier reported results as well as to test on specific devices.

Full system

There are several compile options. Not only does each different device need different compile settings, the libraries used in the implementation do as well. In the Gitlab repository one can find `memory-benchmark.py`[4]. This file runs the makefiles and reads the output to determine memory usage. This is statically computed by the compiler. It reports the total of the entire mTask system, including cryptography.

We decided to not test all devices that mTask supports. This is because it would be a lot of work and the mTask system we build upon does not support all devices. Furthermore, including all devices does not generate more benefit. For devices with

[4]https://gitlab.science.ru.nl/mlubbers/ mTask/-/tree/CryptoBranch

much more memory, it would make little sense to include them, as we do not have to worry at all about memory on those devices. For example, when compiling on the Arduino Mega2560 and ESP8266, it fits easily on the device.

Only the Arduino Uno is included in this experiment. For the ECC library we will test only secp256r1 and secp256k1.

We do not consider the use of the other curves. secp256r1 and secp256k1 are considered. This is because of the key length in the other curves. 160 is already legacy according to NIST [9]. Thus we will consider only the 256 bit versions. The memory benchmark code included can be easily changed to show the other functions as well. Keep in mind that one has to make sure that mTask still compiles correctly with those different settings.

Furthermore we also test with and without the custom square function defined in MicroECC. defining:

```
ECC_SQUARE_FUNC=1
```

Also, we test with different assembly optimizations. MicroECC has three types supported. ASM none means that we will use c code without inline assembly for optimization. This means we rely fully on the compiler to optimize for us. The second option is ASM small, this means that we use the small inline assembly code, optimized to use as little program space as possible. The last one is ASM fast, however during writing the code it turned out that ASM fast has several issues with compiling for some devices. The ESP devices have much more memory than the Arduino devices, thus we did not include them in the test.

The secp256k1, a Koblitz curve, is used by Bitcoin. The difference between secp256k1 and secp256r1 are described by Bjoernsen [5]. In the next paragraph we summarize the paper.

Bjoernsen that the Koblitz curve could be a few bits weaker, but that it does not matter much with the 256 bits. secp256r1 is a curve by NIST. A reason that Bitcoin did not use secp256r1 is because the curve has random numbers in it. It is not clear where those numbers came from. However, other sources list that these numbers are truly chosen randomly[5]. Nonetheless, the wide use of the curve makes it unlikely that the random numbers are chosen for a backdoor. Secp256k1 has so called *nothing up my sleeve numbers*, meaning that it is transparent how they are chosen.

Note however that changing what curve is used in the implementation is as simple as changing one line of code in the server code and changing the compile options for a device.

SHA256, ECC and ChaCha20: separate speed test

Instead of testing only the entire system, we also wish to test individually parts. This is done by isolating the algorithms. This way we can see where the biggest speed bottleneck is. For this part, we only test with the Arduino Mega2560 and the ESP8266. We use the

[5]http://www.secg.org/sec2-v2.pdf

Arduino IDE for this, with a module installed for the ESP8266. We use the non-static branch for Micro-ECC, as it can be installed as a library.

4.3.2 Optimize flash usage

The memory on the devices is very limited. A first step is getting the protocol working, but for some devices there is not enough memory available without optimizing the code further. This research did only a very limited look at optimizing the flash usage.

4.4 Results

In this section we will give the results of the different experiments. We start of with the full system memory usages, after which we will describe each cryptography used separately.

Total memory usages (Full system)

In table 4.2 the total memory of the system is explained for the Arduino Uno. Other devices are excluded. One can easily add more devices in the Gitlab repository[6]

Table 4.2: Arduino Uno memory usage for the full mTask system.

device	curve	sqrt	asm	flash	data
Uno	256r1	0	none	107.4%	97.5%
Uno	256r1	1	none	105.4%	97.5%
Uno	256k1	0	none	102.3%	97.5%
Uno	256k1	1	none	103.6%	97.5%
Uno	256r1	0	small	106.8%	97.5%
Uno	256r1	1	small	104.5%	97.5%
Uno	256k1	0	small	101.1%	97.5%
Uno	256k1	1	small	101.7%	97.5%

The device column specifies which device we run the test on. In our case we tested on the Arduino Uno only, but one may add more devices in `memory-benchmark.py` later on. The curve specifies which curve we use for the signatures. The sqrt column is 1 if we use the optimized square function. The asm column is used to indicate what assembly optimizations we use. Flash memory used is how much memory the device uses in total to store the code. The data is how much SRAM the device uses for the code.

MicroECC speed performance

We get an average of around 0.57 seconds over 20 runs on the ESP8266, using the secp256r1 curve.

[6]https://gitlab.science.ru.nl/mlubbers/mTask/-/tree/CryptoBranch

Table 4.3: SHA256, bytes processed and speed. The bytes are processed at once.

Bytes	Mega256	ESP
32	11 (ms)	0.09 (ms)
64	21 (ms)	0.17 (ms)
256	53 (ms)	0.4 (ms)
2048	351 (ms)	3 (ms)

Table 4.4: ChaCha20 encryption/decryption on the Arduino Mega2560.

nr Bytes	Time
10	3216 μs (3ms)
100	6439 μs (6 ms)
120	6466 μs (6 ms)
4096	200 (ms)

We get an average of around 5.18 seconds over 20 runs on the Arduino Mega2560, using the secp256r1 curve and the non-static branch.

For the Arduino this is around twice as slow as reported on MicroECC static branch.[7]

SHA256 speed performance

SHA256 can update in different ways. One could update SHA256 one byte at a time, or send a big message at once. The difference is very small. On the Mega, 2048 bytes takes 366 ms with doing it one byte at a time and 351 ms if done at once. For smaller sizes, e.g. most tasks, it differs less then one millisecond.

The library we used also tested the performance with similar results [8].

ChaCha20 speed performance

The ESP8266 had memory alignment issues with compiling using the Arduino IDE and is thus excluded in the results. Furthermore, ChaCha20 uses 20 rounds and setting the IV/key is included in the results. For this, a 256 bit key used. It takes around 16 μs on the ESP8266 and around 100μs on the Arduino Mega to just set the IV and key. Because ChaCha20 is a stream cipher, calling the decryption function results in a direct call to the encryption functions and thus has similar performance.

At first glance, this is much slower than described by the library we use[9]. However, keep in mind that the performance is in multiple of 64 bytes. After each 64 bytes we compute the next block.

[7]https://github.com/kmackay/micro-ecc/tree/static.

[8]https://rweather.github.io/arduinolibs/crypto.html

[9]Arduino Uno performance: https://rweather.github.io/arduinolibs/crypto.html

Chapter 5

Related Work

There are several categories within cryptography. There are papers about public key crypto, symmetric crypto, (secure) hash functions, Message Authentication Code functions (MAC) and random number generators. Lastly, there are encryption schemes that combine those to offer one or more security requirements. We will describe different existing research about the algorithms, as well as some encryption schemes used for IoT.

Public Key Cryptography (PKC)

Zhang et al. describe ongoing challenges for each of the security principles we described in Chapter 2 and other challenges in IoT security [23]. It is mainly focused on public key cryptography. One of the key problems is that the devices cannot be trusted, a shared key can be assumed to be compromised at some time. In some applications this is a good compromise, but we assumed that the IoT devices are not very secure.

In a further section of Zhang's paper some crypto schemes are explained that use some sort of PKC. In general, the performance costs of PKC seems to be too high for use in all communication of the mTasks system. In ECDSA on things [3] there is a list how they did digital signature authentication on IoT devices. The authors present a solution for the slow signing. Instead of signing every message, it only signs a message once every 5 seconds. The paper did not succeed in finding a solution for a device with only 8KB RAM and around 60KB flash memory. They did however managed to implement it for the 32bit CC2538 chip, which has 32KB RAM.

Symmetric Key Cryptography (SKC)

Alassa et al. give an overview of lightweight symmetrical key cryptography [2]. In the paper an overview is presented of the clock cycles used per byte of encoded data as well as memory used. It appears that symmetric key cryptography offers reasonably fast encryption of data. Note that SKC does not in itself provide any other security requirement such as integrity.

The webpage of CryptoLUX[6] shows the performance of different cryptography primitives. It describes a small MAC for IoT devices and is deemed very suitable for our need. Chaskey [18] and AES [7] are shown as best in the comparison for small code size and RAM.

Instead of using a dedicated stream cipher, it is possible to build a stream cipher from a block cipher. This way we do not need the decryption function of the block cipher. Another benefit is that we do not need padding. It is however crucial to make sure that there is no nonce violation. Moreover, some block ciphers have an even harder requirement for the IV. For example, a block cipher in CTR mode must make sure that it has random initialization vectors (IVs) in order to have confidentiality.

Encryption schemes

Liu et al. describe existing authentication and access control methods in their research paper [15]. They make it feasible for IoT. Just like the mTasks system it uses one authority, the server. The proposed protocol for authentication only is already quite elaborate. The protocol allegedly can prevent eavesdropping, man-in-the-middle, key control attacks and replay attacks. According to the paper it uses a PKC-based scheme, which suffers from high energy consumption and considerable time delay.

He and Zeadally show heavyweight, middleweight and lightweight Elliptic curve cryptography schemes in their paper [11]. For each of the schemes it is shown what security requirements are met. Although a lot of possible schemes are described, the computational cost is still too high.

Keoh et al. describe a new scheme, DTLS [13], which is similar to TLS. The total RAM used (state machine, crypto, key and DTLS record layer) is 3.9KB. It is not as suitable for mTask because of the complexity and the resources it uses.

Hammi et all describe how a wireless sensor network can use a security protocol that is lightweight and secure [10]. The data is encrypted and authenticated. It does however use symmetric cryptography only, making it not suitable for our attackers model.

Sadio et al. propose an MQTT solution that uses chacha20-poly1305 [20]. chacha20-poly1305 is AEAD, which means that it also authenticates the data. It works for the Arduino Uno. The AEAD version for Arduino uses 654 bytes RAM. Encrypting or decrypting 8 bytes takes 2-2.5 seconds. 64 byte takes a bit over 3.5 seconds. This research is very promising, but it is already a bit on the computational intensive side.

Chapter 6

Conclusion and discussion

In this thesis we aimed to provide a lightweight cryptography protocol for the use in the communication channels for IoT devices. Especially, we aimed to build such a solution for the mTask system. We were able to provide mTask with a more secure version for communication of tasks. The performance is very reasonable for the encryption and hashing.

The biggest speed penalty and also the memory penalty, is in the MicroECC library. Taking our own data into account, the theoretical time of signing a relatively sized task, around 256 bytes, takes between 5-6 seconds. A few seconds is not very fast for a modern computers, but for IoT devices this time is often very reasonable.

The best parameters to use for the system is curve secp256k1, using small assembly with the separate (faster) squaring function disabled. This full system does not yet fit in the Arduino Uno memory, as it needs 101.1% of the memory.

Signatures provide a way of authentication, but with slight adjustments, one could use HMAC instead of signatures, for example for SDS and other less important messages like the device specs. This should improve timing much further.

The target code size is very close the one we achieved. Only barely not fitting on the Arduino Uno board. With a little bit of effort we think that it is feasible to get it working on the Arduino Uno.

During experimenting, we were not able to get the static MicroECC branch working on the ESP8266 within reasonable time. This was due to issues with makefile for ESP. Instead, we decided to use the non static branch for testing. To still have a fair comparison, we decided to use the non static version for Arduino as well during testing the ECC speed performance. This provided results twice as slow as reported by the library used itself. Nonetheless, it gives a reasonable expectation of speed. It will not be very fast on the restricted devices, but not very slow either depending on the usages.

More surprisingly, the ESP8266 did not perform as well as initially thought. Having a 32 bit architecture as well as having a clock speed 5 times as high a the Arduino Mega, it only performed 10 times as fast. This could be due to the MicroECC library not having optimizations for the ESP8266,

Chapter 7

Future research

There are still several parts that can be research much further. This research did not focus on the availability of the devices. It may be possible to improve a lot on this area. Another improvement could be to improve the memory and computational resources used by the algorithms used in this research.

One of the most important improvements is to add the security to more messages in the communication channel. Currently, only tasks are using the security module. Implementing the other other communication should be straightforward, as the communication is very similar to the task communication. One could research if using a HMAC instead of a signature for other communication messages may be faster and still secure enough. Note that using asymmetrical cryptography for sending the tasks is still a requirement.

Furthermore, the codebase could be improved much further. Things such as maintainability needs to be improved. Storing keys and nonces on the server or device for use after a power outage or reset is not implemented. The key is relatively small and to mitigate for the limited number of writes the EEPROM can sustain, cycling through memory is a feasible way to store the key and nonce.

Additionally, error handling is not yet done well. This is mostly related with the availability aspect. Not every error is sent to the server.

Lastly, more research into the area of asymmetric key cryptography could help a lot with improving the speed of cryptography schemes on IoT devices.

Bibliography

[1] Peter Achten, Pieter Koopman, and Rinus Plasmeijer. *An Introduction to Task Oriented Programming*, pages 187–245. Springer International Publishing, Cham, 2015. `doi:10.1007/978-3-319-15940-9_5`.

[2] Norah Alassaf, Basem Alkazemi, and Adnan Gutub. Applicable light-weight cryptography to secure medical data in IoT systems. *Journal of Research in Engineering and Applied Sciences (JREAS)*, 2:50–58, 04 2017.

[3] Johannes Bauer, Ralf C Staudemeyer, Henrich C Pöhls, and Alexandros Fragkiadakis. Ecdsa on things: IoT integrity protection in practise. In *International Conference on Information and Communications Security*, pages 3–17. Springer, 2016.

[4] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The simon and speck families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. `https://eprint.iacr.org/2013/404`.

[5] Kristian Bjoernsen. Koblitz curves and its practical uses in bitcoin security. *order (ε (GF (2k)*, 2(1):7, 2009.

[6] Cryptolux. Lightweight cryptography. URL: `https://cryptolux.org/index.php/Lightweight_Cryptography`.

[7] Joan Daemen and Vincent Rijmen. *Rijndael/AES*, pages 520–524. Springer US, Boston, MA, 2005. `doi:10.1007/0-387-23483-7_358`.

[8] Christophe De Canniere and Bart Preneel. Trivium specifications. In *eSTREAM, ECRYPT Stream Cipher Project*. Citeseer, 2005.

[9] Damien Giry. Nist report on cryptographic key length and cryptoperiod (2020). URL: `https://www.keylength.com/en/4/`.

[10] M. T. Hammi, E. Livolant, P. Bellot, A. Serhrouchni, and P. Minet. A lightweight iot security protocol. In *2017 1st Cyber Security in Networking Conference (CSNet)*, pages 1–8, Oct 2017. `doi:10.1109/CSNET.2017.8242001`.

[11] D. He and S. Zeadally. An analysis of rfid authentication schemes for internet of things in healthcare environment using elliptic curve cryptography. *IEEE Internet of Things Journal*, 2(1):72–83, Feb 2015. doi:10.1109/JIOT.2014.2360121.

[12] Nicholas Jansma and Brandon Arrendondo. Performance comparison of elliptic curve and rsa digital signatures. *nicj. net/files*, 2004.

[13] S. L. Keoh, S. S. Kumar, and H. Tschofenig. Securing the Internet of Things: A standardization perspective. *IEEE Internet of Things Journal*, 1(3):265–275, June 2014. doi:10.1109/JIOT.2014.2323395.

[14] Benedikt Kristinsson. Ardrand:the arduino as a hardware random-number generator. *Reyjavik University*, 12 2011.

[15] J. Liu, Y. Xiao, and C. L. P. Chen. Authentication and access control in the internet of things. In *2012 32nd International Conference on Distributed Computing Systems Workshops*, pages 588–592, June 2012. doi:10.1109/ICDCSW.2012.23.

[16] M. Lubbers. Task oriented programming and the Internet of Things. *Radboud University*, 6 2017.

[17] Rwan Mahmoud, Tasneem Yousuf, Fadi Aloul, and Imran Zualkernan. Internet of things (iot) security: Current status, challenges and prospective measures. In *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, pages 336–341. IEEE, 2015.

[18] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: An efficient mac algorithm for 32-bit microcontrollers. In Antoine Joux and Amr Youssef, editors, *Selected Areas in Cryptography – SAC 2014*, pages 306–323, Cham, 2014. Springer International Publishing.

[19] Helena Pous and Jordi Herrera-Joancomartí. Computational and energy costs of cryptographic algorithms on handheld devices. *Future Internet*, 3, 12 2011. doi:10.3390/fi3010031.

[20] O. Sadio, I. Ngom, and C. Lishou. Lightweight security scheme for mqtt/mqtt-sn protocol. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 119–123, Oct 2019. doi:10.1109/IOTSMS48152.2019.8939177.

[21] Y. Shafranovich. Hmac: Keyed-hashing for message authentication. RFC 2104, RFC Editor, 2 1997. URL: https://tools.ietf.org/html/rfc2104.

[22] Zeinab Vahdati, Ali Ghasempour, Mohammad Salehi, and Sharifah Md Yasin. Comparison of ecc and rsa algorithms in iot devices. *Journal of Theoretical and Applied Information Technology*, 97:4293, 08 2019.

[23] Z. Zhang, M. C. Y. Cho, C. Wang, C. Hsu, C. Chen, and S. Shieh. IoT security: Ongoing challenges and research opportunities. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pages 230–234, Nov 2014. `doi:10.1109/SOCA.2014.58`.