Benjamin Kuter, Niantong Dong
EC544 - Technical Report

Introduction

Wow, what a project! We originally struggled with setting up and understanding the K64F environment, but learned a lot by the end of the project. At the end of the day, however, we have successfully created a program, loaded onto the K64F, that takes a user-supplied message (in this case it can be a bitcoin transaction) and hashes, signs, and encodes it. For testing purposes, the program can also decode the message from base64 and verify it using our public key. To start off our discussion, we will immediately jump into the shortcomings and where improvements could have been made. Then we'll describe the bitcoin transaction structure and our transaction model. Afterwards, we'll delve into the more exciting technical aspects of the project, where we got stuck, and how we resolved those roadblocks!

Shortcomings & Difficulties

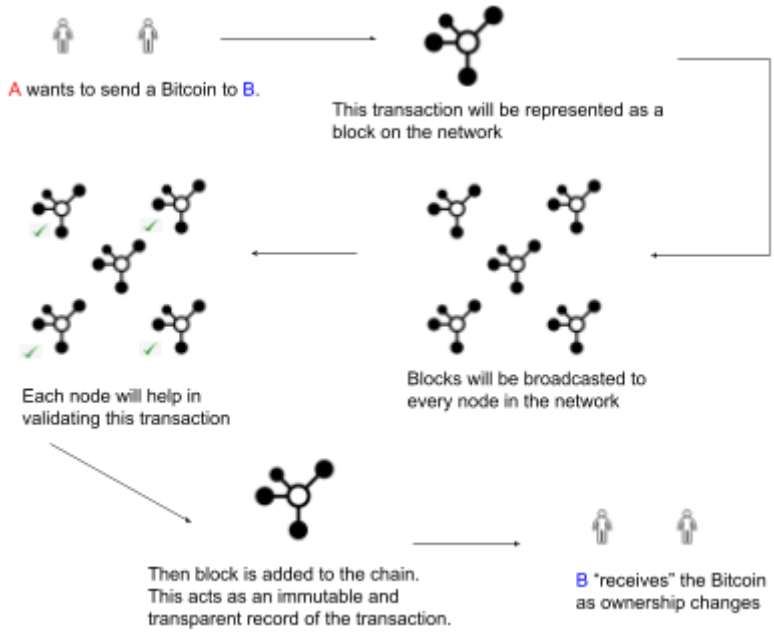
First, let's make our obligatory "COVID-19" shoutout. The added difficulties due to remote communication were a mild hindrance, but luckily it is 2021 and we have somewhat adapted to COVID conditions. Online meetings have become the norm and made debriefs very easy. Sometimes it was difficult to code as a team, but screen sharing alleviated much of that stress. Because of the very programmatic nature of our project, we didn't need anything in the way of sensors and all necessary work could be shared easily over GitHub. The IDE could be a bit finicky at times, as the K64F would sometimes become unbootable and would need to be reflashed with Segger's JLink firmware. This happened often enough to be an annoyance during development but never seriously hindered the project.

There were several grand features in our original design that were never implemented due to the realistic scope of a semester-project. We are still very satisfied with our overall design and progress over the semester, but features such as encrypting the device, file I/O for handling separate RSA key files, and API calls for generating Bitcoin transactions had to be forgotten. Creating a bootable device with all the necessary supporting code proved to be more difficult than expected. We had to go through a number of demo codes before we found one that worked. Specifically, we were having trouble initialising our four mbedtls contexts. This was eventually resolved by using the mbedtls demos instead of the AWS demos. A lot of code had to be pruned, but all of the necessary mbedtls infrastructure was already in place. A final complication was the lack of a *real* Bitcoin transaction. Neither of us own actual cryptocurrency and we were unable to acquire any for this project. We tried using BTC testcoins, but these didn't work in the transaction-generators we used. To remedy this, we made faux transactions (which we will cover later) that we passed into our transaction-generator. Of course, these were not pushed to the blockchain.

There are definitely improvements to be made, especially if this were to be deployed as a full product, but that's why product design & development takes years with experienced teams. For the time available, and lack of preexisting knowledge, we believe we have a very technically-interesting gadget to show for our work.

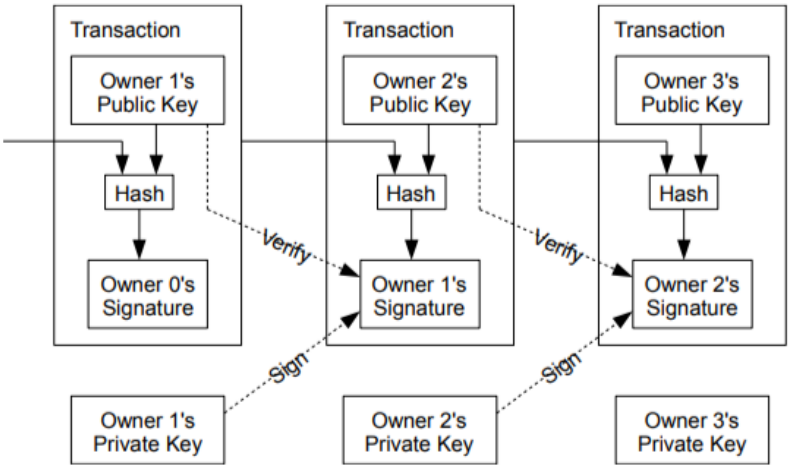
Bitcoin Transactions

In this section, we want to talk about how bitcoin works and why people think this technology is able to change the world.



To learn more about how bitcoin works, we need to first learn more about blockchain. A blockchain is a database where it is practically impossible to hack or cheat the system. The network is programmable, secure, anonymous, unanimous, time-stamped, immutable, and distributed.

A bitcoin transaction is how we exchange currency in the network. A transaction is a transfer of Bitcoin value that is broadcast to the network and collected into blocks. The way transactions are documented in the distributed database through mining is a key innovation of Bitcoin. Transactions are organized into blocks, and around every 10 minutes, a new block of transactions is sent out, becoming part of the blockchain. These serve as official records of the transactions. New bitcoins are now added to the scheme by mining. When a block is successfully mined, it generates new bitcoins and pays them to the miner. In addition, the miner receives any payments involved with the block's transactions. As a result, mining is extremely competitive, with a large number of people attempting to mine blocks. Mining complexity and competition are important aspects of Bitcoin security because they mean that no one can overwhelm the blockchain with bad blocks through sheer computational force.



Nakamoto, Satoshi. (2009). Bitcoin: A Peer-to-Peer Electronic Cash System.

This diagram depicts how contracts are signed and joined together in a condensed manner. Consider a middle exchange, in which bitcoins are transferred from address B to address C. The contents of the transaction are hashed and authenticated with B's private key, including the hash of the previous transaction. In addition, the exchange includes B's public key. Anyone can check that the transaction is approved by B by following a series of steps. To begin, B's public key must match B's address from the previous transaction, demonstrating that the public key is correct. The public key can easily be used to calculate the address. The transaction's signature will then be checked using B's public key in the transaction. These measures ensure that B has approved and validated the transaction. B's public key isn't made public until it's used in a trade. Bitcoins are exchanged from address to address in a series of

transactions using this method. To guarantee that bitcoins are being spent correctly, each move in the chain can be checked. Since transactions in general may have several inputs and outputs, the chain grows into a tree.

Our Faux Bitcoin Structure

To approximate a bitcoin transaction, we utilised coinb.in's transaction generation feature. First we made a SigWit Address (*bc1qnqn4n4pmghxz84aupa2n8n60prjeknld0gu2sw*) to act as our destination address. Then we investigated the current block chain through blockchain.com, looking for an address that currently had an UTXO (unspent transaction output), resulting in address *bc1q2wraskwtlreqrk2uk6d9mqqcgqtj2yxpe8tjf4*. We needed to do this to trick coinb.in to generate a transaction. We did not attempt to actually use this transaction on the blockchain, and would be unable to without the original recipient's private key. With our input and output address, we set our transaction amount to 0.0008 BTC and generated our raw BTC transaction. At the time of writing, this was:

```
0100000002a14d86e95360199262339b9d6186a4a54b399f4c825497117eb9c12f82baaef5000000001f145387d859cbf8f20
1d95cb69a5d801840172510c1000896cd000000000000fdffffff6edb840a0c7c88474495f4646977f3d538f2b89e38185f4f
9b1c84e08f994f49000000001f145387d859cbf8f201d95cb69a5d801840172510c10008448700000000000fdffffff01803
80100000000001600149827 59d43b45cc23d7bc0f5533cf4f08e59b4fed00000000
```

This raw transaction was hardcoded into our wallet, and used as our to-be-hashed message. Inside of this value, we see the following scripts corresponding to the components of the input address:

```
145387d859cbf8f201d95cb69a5d801840172510c100084487000000000000
145387d859cbf8f201d95cb69a5d801840172510c1000896cd000000000000
```

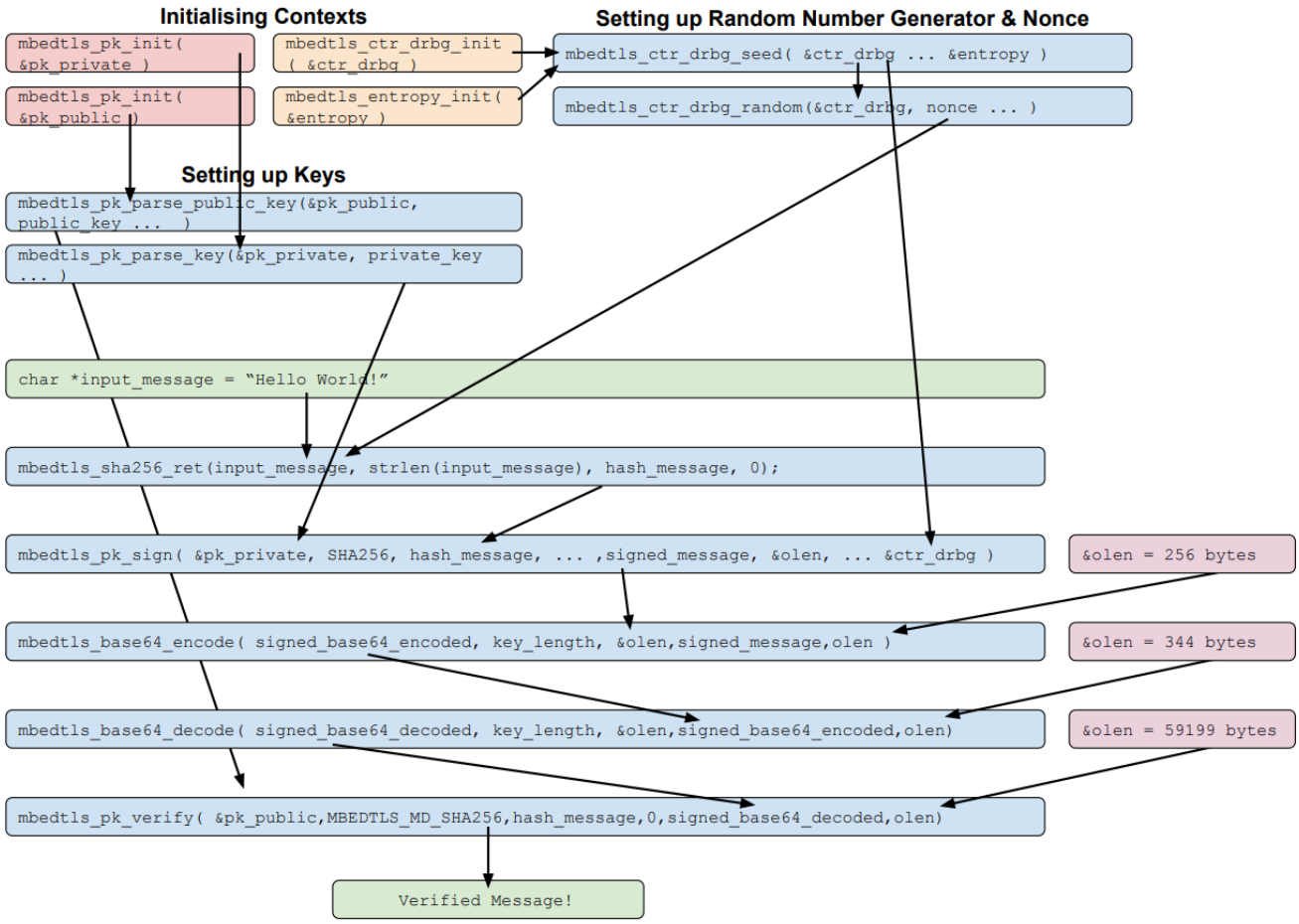
These hex sequences are the transactions that make up our input transaction, which can trail recursively up to some original transaction. And with that, on to the achievements and technical description!

Technical Description

Let's review the overall goal of our device and code. We want to take a user provided message, hash it, and then sign it with our private key. This process offers non-repudiation and helps ensure both data integrity and message authentication. The hashing solves the important message-length issue with keys. Our keys were generated with RSA2048, which has a maximum length of 2048 bits (256 bytes), minus padding. Messages can very easily be longer than that, so we ensure that every message has a fixed 32 byte width using SHA256. From here we want to present and transmit our signed message in a more human-readable form, so we base64 encode. This step is necessary because many platforms expect data to be represented as ASCII, not binary. At this point, our primary goal is done. For this project, however, we want to present the full picture for validation's sake. After encoding and sending our signed message out, we also decode it and then verify it using our public key. Our system is built upon an mbedtls cryptography testing example, and as such utilises many of their cryptographic functions. This open-source SSL library is used by FreeRTOS for its TLS authentication[1,2].

Now,a more technical description of the major sections of our code, along with major troubleshooting issues. Our general code stack is as follows:

[1] https://tls.mbed.org/about-us
[2] https://docs.aws.amazon.com/freertos/latest/portingguide/afr-porting-tls.html

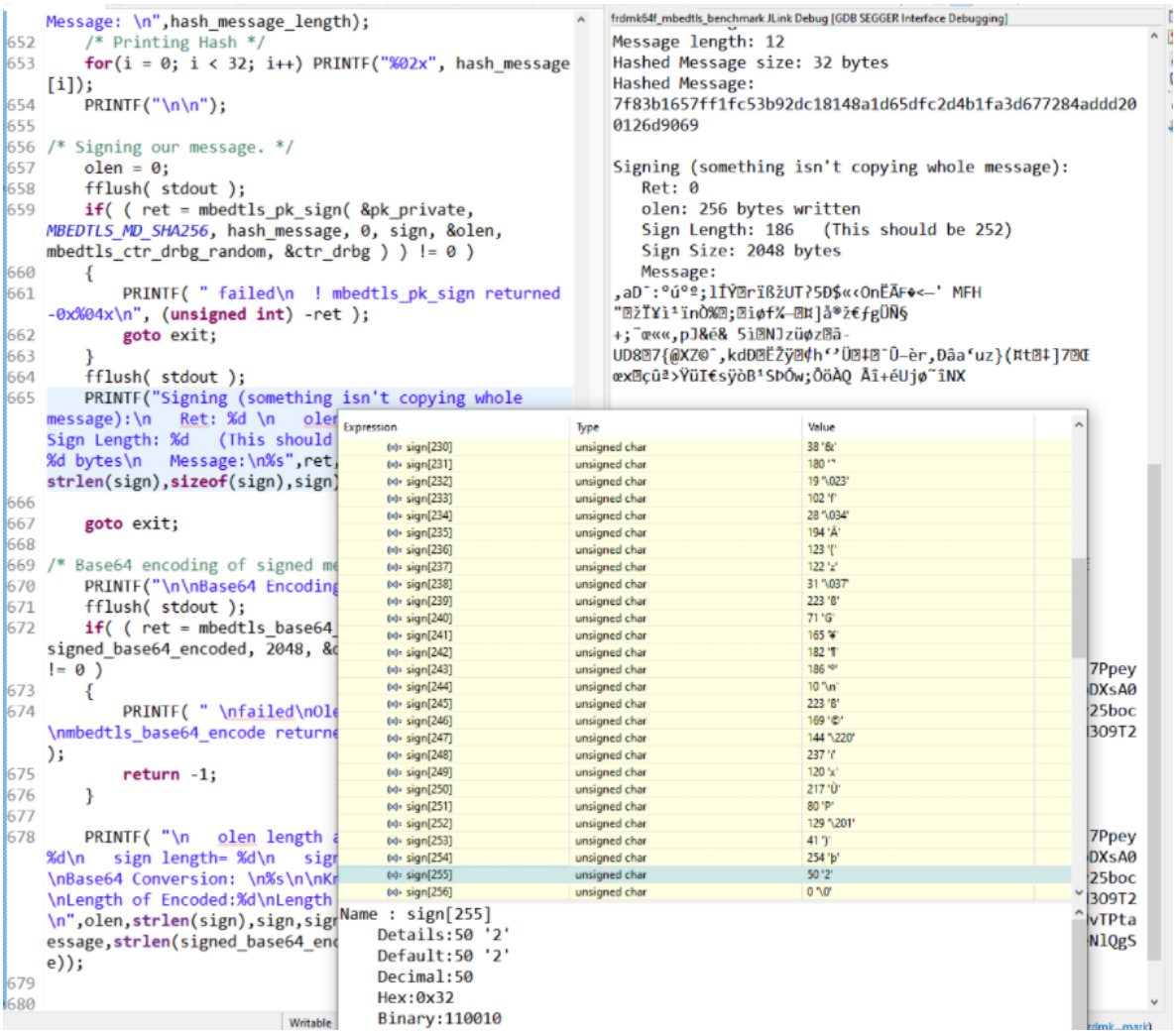


The `input_message` is a proxy for any type of input, which for us will be a raw bitcoin transaction. A `nonce` is generated using a random number generator, which is setup through the `ctr_drbg context`. This is appended to the input message and here acts as a proxy for the nonce typically solved by blockchain miners. The context initialisation sets up the mbedtls structures for later use. The public and private key pairs are then set up using a hardcoded string, each containing the keys respective value. These are later used for signing and verifying the hashed message.

At this point we had extensive issues with our signed message truncating after ~186 characters, resulting in our verification process failing. As the starting public and private keys were generated with the RSA2048 standard, with fixed lengths of 256 bytes, if one were to do `strlen(signed_message)` they would expect to return 256 bytes. This was not the case, as it was returning 186 bytes. If you immediately know what we were doing wrong, kudos to you, but figuring out why took a considerable amount of time (around seven days) for us. We needed to consider what the output of the signed message actually looked like on a bit-level.

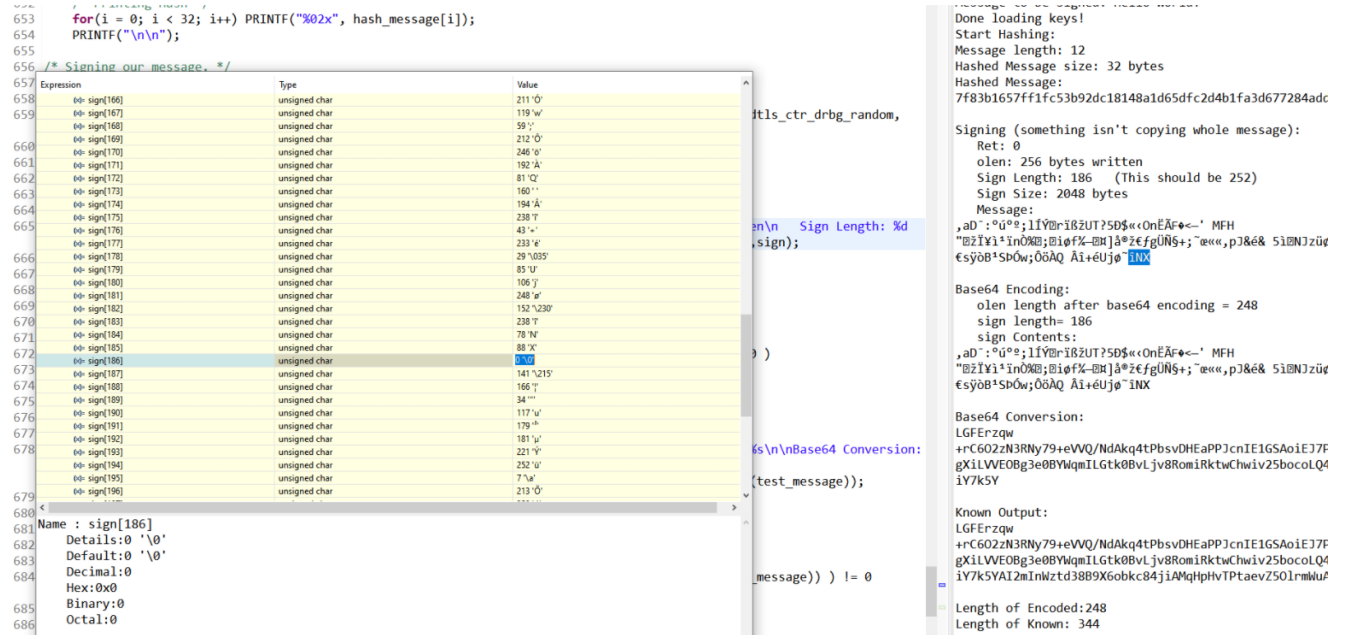
Altering the message on the bit-level seems like one of the most secure approaches, as the bits are practically at the smallest layer we can change. Because of the mbedtls functions we used we had to still handle these messages as `char*` (`strings`). This data type lets us use some fun tools, like `sizeof(array_head)` to tell us the size in bytes each array element if the string is dynamically allocated (`malloc()`, `calloc()`). What's interesting about `sizeof()` is that on statically declared arrays, like `MyArray[2048]`, `sizeof(MyArray)` will tell us the entire length of the array. Cool!

So `sizeof(signed_message)` was 186 bytes, but we also knew that 256 total bytes were actually written into the string. For some reason these remaining elements weren't accessible, as `printf("%s",signed_message)` would truncate the string. We knew the signing had worked, as an online comparison showed the first 186 characters were identical. If there had been any downstream issues, our signing would be very different given the hashed nature of our input message. So we knew we had a buffer issue where we weren't able to read the entire contents of the file. Why?



When we inspect `char * sign[]`, our signed hashed value, it does indeed have 256 elements which seem to line up with what we expect, but it's only printing 186 of them. This tool is one of our favourite MCUXpresso features thus far. We had to insert a breakpoint at the start of the exit condition in main(), but that's normal troubleshooting. Hovering over a variable during runtime while paused at the breakpoint let us inspect array elements in real-time, helping validate not only the size of the signed message, but also confirming that it was the same as the web-calculated version.

Now we must take a closer look at where exactly our message was truncating:



Well, would you look at that! What we find right after `îNX` in the raw string is a `\0` and it just so happens to be at element 186, exactly where our `strlen()` function was returning. That would definitely be our issue. The binary output of the signed message was being cast as a character in our string. At some point in that bitstream, the sequence `00000000` aligned with the start of a new character in the character array, which was then interpreted as a `\0`, the null character. Remember the signed message is just some encrypted value, not representative of any actual characters. It has to be decoded first, otherwise it's meaningless junk.

By changing the length arguments used in our mbedtls functions to `&olen` from `strlen(message)`, and implementing different print methods for the encrypted base64 messages, we were able to print the expected results *sans truncation*. From here all that was left was base64 encoding our hashed-signed message as it is generally easier and more portable to use ASCII characters than binary values. Using `&olen` from here on out was crucial. The base64 decoding segment worked very similarly to the base64 encoding step and returned expected results once `&olen` was implemented. Finally the verify segment also worked properly once the correct `sig_len` argument was used. In a deployed solution, the original nonce and transaction would be sent at a later point, allowing a third party to verify our message some time after it was originally transmitted/received.

Takeaway Skills

Two technical skills developed over this project immediately spring to mind. Neither of us went in knowing much about cryptography, and even less about developing on an embedded systems platform. What became abundantly clear was the importance of reading the developer's documentation thoroughly. The clear and precise API descriptions provided by arm-MBED were absolutely essential. The function references were very well explained and the included demos were great for understanding how segments of code were operating. This experience of delving through documentation is definitely the most important takeaway from this project. A reputable API will provide accessible documents and demos, regardless of its complexity. Many coding problems can be solved by just spending ample time reviewing documentation and planning out the control flow. In this ever-growing world of device

interconnectivity, accurate authentication and authorization is critical. Learning these cryptographic skills while simultaneously improving our coding prowess will be beneficial in our future embedded careers.

Sources

- https://medium.com/the-mission/a-simple-explanation-on-how-blockchain-works-e52f75da6e9a
- http://www.righto.com/2014/02/bitcoins-hard-way-using-raw-bitcoin.html
- https://bitcoin.org/bitcoin.pdf
- https://www.quora.com/Does-Bitcoin-run-on-Blockchain
- https://tls.mbed.org/discussions/generic/parsing-public-key-from-memory
- https://tls.mbed.org/kb/how-to/encrypt-and-decrypt-with-rsa
- https://www.cs.cornell.edu/courses/cs5430/2015sp/notes/rsa_sign_vs_dec.php#:~:text=To%20sign%20a%20message%20m,result%20equals%20the%20expected%20message.&text=That's%20the%20textbook%20description%20of,or%2Dless%20the%20whole%20story.
- https://tls.mbed.org/api/base64_8h.html
- https://tls.mbed.org/api/pk_8h.html#a072d27dc4143bfd600786232f9417e08

-