Informatik Projekt 2

Post Quantum Cryptography on Raspberry Pi

12mojo1bif@hft-stuttgart.de
12test11bif@hft-stuttgart.de
12test21bif@hft-stuttgart.de

Group: 4

Group Members:
Jonas Möwes, Ayham, Omar

2025-02-02

Contents

1 Introduction

In this project we explored the efficency of Post Quantum Cryptography (PQC) Algorithms on Microcontroller. Therefore we build a benchmark that tests choosen algorithms with real data. We also wanted to include a completly different implementation that is written in Rust instead of C.

In the beginning we tried to run these algorithms on very lightweight systems like an ESP32 or even an ESP8266. See (Doc) for further Information.

This document will set the Focus on what the results of the finished project accomplished.

After we realized that it wont be as easy as thought to run PQC algorithms on these very lightweight systems we decided to focus on the Raspberry Pi implementation. The implementation of the benchmark changed alot over the time of this project but always had the goal to create a real scenerio on how a embedded system would publish its data to a server while this beeing Post Quantum safe. The Realisation section goes over some mid level implementations.

Broken down to the communication of the server and client we wanted to achieve a communication like in Figure 1:

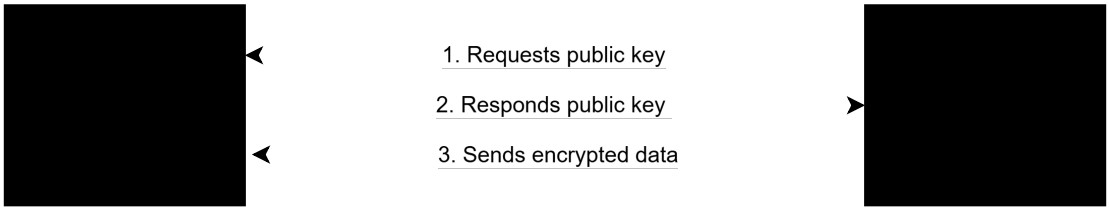

Figure 1: Communication of Server and Client

As implementations of the Cryptography algorithms we used the PQClean library. For the Rust implementation we used the library kyber from Argyle-Software.

PQClean is a library providing clean, portable, and secure implementations of post-quantum cryptographic algorithms. It focuses on quantum-resistant encryption, key exchange, and digital signatures, following strict coding guidelines for security and auditability. The library includes implementations of NIST PQC standardization candidates while ensuring resistance to side-channel attacks. Written in standard C, it is designed for portability and ease of integration into security-critical applications. PQClean serves as a trusted foundation for developing and benchmarking post-quantum cryptography.

2 Realisation

2.1 General Setup

As Hardware we had at the start a Raspberry Pi Zero 2 W and a Raspberry Pi 3B. Later we got a second Raspberry Pi 3B.

Due to much capabilities of the Raspberry Pis we were able to run a simple Python Flask server and make requests to it with the requests library of Python.

Here is a list of which packages are needed:

```
pycryptodome
flask
requests
paramiko
```

```
pandas
python-dotenv
```

To be able to run the in C written Algorithms of the PQClean library we used ctypes from python. To do this we needed to compile the C-Files. To make this is automated with python scripts that creates a Makefile for each Algorithm and runs make with it. The result is a build directory with subdirectory crypto_sign and crypto_kem which contain the binarys of the algorithms. Because PQClean implementations have a guidline on the interface it was pretty easy to call them. The function headers for the Key Encapsulation Mechanism are always:

```
  int PQCLEAN_NAME_CLEAN_crypto_kem_keypair(uint8_t *pk, uint8_t *sk);
  int PQCLEAN_NAME_CLEAN_crypto_kem_enc(uint8_t *ct, uint8_t *ss, const
uint8_t *pk);
  int PQCLEAN_NAME_CLEAN_crypto_kem_dec(uint8_t *ss, const uint8_t *ct,
const uint8_t *sk);
```

For the kyber512 algorithm it is:

```
int PQCLEAN_KYBER512_CLEAN_crypto_kem_keypair(uint8_t *pk, uint8_t *sk);
int PQCLEAN_KYBER512_CLEAN_crypto_kem_enc(uint8_t *ct, uint8_t *ss, const
uint8_t *pk);
int PQCLEAN_KYBER512_CLEAN_crypto_kem_dec(uint8_t *ss, const uint8_t *ct,
const uint8_t *sk);
```

The function headers for the Signature Mechanism are always:

```
int PQCLEAN_NAME_CLEAN_crypto_sign_signature(
    uint8_t *sig, size_t *siglen,
    const uint8_t *m, size_t mlen, const uint8_t *sk);

int PQCLEAN_NAME_CLEAN_crypto_sign_verify(
    const uint8_t *sig, size_t siglen,
    const uint8_t *m, size_t mlen, const uint8_t *pk);

int PQCLEAN_NAME_CLEAN_crypto_sign(
    uint8_t *sm, size_t *smlen,
    const uint8_t *m, size_t mlen, const uint8_t *sk);
```

For the dilithium2:

```
int PQCLEAN_DILITHIUM2_CLEAN_crypto_sign_signature(
    uint8_t *sig, size_t *siglen,
    const uint8_t *m, size_t mlen, const uint8_t *sk);

int PQCLEAN_DILITHIUM2_CLEAN_crypto_sign_verify(
    const uint8_t *sig, size_t siglen,
    const uint8_t *m, size_t mlen, const uint8_t *pk);

int PQCLEAN_DILITHIUM2_CLEAN_crypto_sign(
    uint8_t *sm, size_t *smlen,
    const uint8_t *m, size_t mlen, const uint8_t *sk);
```

The Rust implementation has a different header. Therefore there is a wrapper in the rust directory. To compile this wrapper and to create the needed binarys there is a shell script to build them. They are also moved into the *build/crypto_kem* directory. (See Furthere Notes for Rust installation.)

2.2 First implementation

The first implementation we finished was quite different to what is the end result. Figure 2 shows the communication of the server and client in this case.

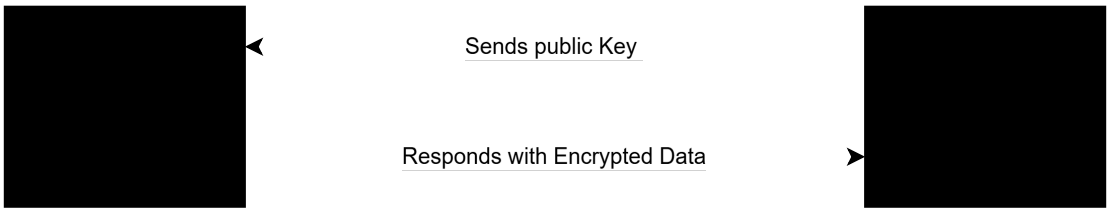

Figure 2: First implementation

Here the Server receives a public key for an key encapsulation mechanism and returns the encrypted Data. The server also signs the hash of the encrypted content. In this process it benchmarks various process steps like performing the key encapsulation mechanism, encrypting the data, hashing the encrypted data and signing the hash. These measurements are also send back as the response to the client. The client then verifies and decrypts the response, while also measuring the time. Then it saves all the timeings to a file.

This is benchmark almoste achieved what this project was up for. But the client/server role was off. In this scenario the server is the one that is measuring for example the temperature and then gets asked by the client to sends it to him. This is totally fine but would mean that every device would have to run a server. It would be preferable (from a real world perspective) to have the device that is measuring the temperature to be the client and the device that is collecting the data to be the server.

2.3 Implementation

Figure 1 shows the communication of the server and the client in the finished implementation. The client requests the public key of a key encapsulation mechanism. With this key it creates an AES key and encapsulates it. This AES-Key is then used to encrypt the data. The encrypted data is then hashed and then signed with a post quantum signature mechanism. Every process is measured and saved to csv file.

The client then sends the encrypted data, the signature, the public signature key, the used sign and kem algorithm, the encapulated AES-Key and the IV to the server.

The server hashes the encrypted data to then verify the signature. It then decapsulates the AES-Key and decrypts the data. Every process is also measured and written to a csv file.

This is the underlying Server and Client functionality of this benchmark. It also needs to be said that when starting the server and client they write the time to generate the public and private keys for each algorithm into a csv file.

To now run this benchmark you could install setup the server on a device A and the client on a device B. Then start the server on device A and then start the client on device B. This would then run until you stop the client.

2.4 Controller

As mentioned the client and the server only once (at the start) generate their keys. So it would be nice to restart server and client to also compare the key generation times of the algorithms. Also especially when using two different devices it would me nice if the server and client role of the device would swap.

You could now do this by hand by restarting the server and client in an intervall. But creating the keys on these lightweight devices takes a lot of time (See Evaluation) so this is not recommended.

Instead there is a Controller script that uses ssh and scp to manage the benchmark and orchestrate the files created by the server and client. This controller script depends on the way your devices are set up. To use it you need to specify the host name and the location of your ssh keys establish an connection with the devices. In the controller script you can set the nbumber of iterations and the time for each iteration.

At the start of the script the controller deletes all csv file in the directory on the devices so be carefull.

When finished with the benchmark it downloads all created files and puts them together.

This creates the following CSV-Files:

```
client_timings.csv
server_timings.csv
key_generation_times.csv
```

3 Further Notes

Here are some minor details that are important to run the benchmark correctly.

- .env files: Create .env files in the directory like this:

```
DEVICE_NAME=<Device name>
```

this how the server and client get the name to write to in the CSV-Files.

- The controller script uses an shell script to make sure everything is installed. If not it tries to install everything and then runs the benchmark. Should there be an error in the installation process it wont run the benchmark.
- The controller script will run the complete installation process. This includes the creation of the binaries from the PQClean library and the Rust wrapper.
- The controller script uses ssh keys to authenticate. This seems not to work with rsa keys. `ssh-keygen -t ed25519` works.

4 Evaluation

5 Conclusion