Informatik Projekt 2

Post Quantum Cryptography on Raspberry Pi

12mojo1bif@hft-stuttgart.de
22alay1bif@hft-stuttgart.de
12haombif@hft-stuttgart.de

Group: 4

Group Members:
Jonas Möwes, Ayham Alhasan, Omar Haj Abdulaziz

2025-02-03

Contents

1 Introduction

This project explores the efficiency of Post-Quantum Cryptography (PQC) algorithms on microcontrollers. To achieve this, a benchmark was developed to test selected algorithms using real data. Additionally, a completely different implementation written in Rust, instead of C, was included for comparison.

Initially, the goal was to run these algorithms on extremely lightweight systems, such as the ESP32 or even the ESP8266. More details on this initial attempt can be found in (Doc).

This document focuses on the final results and accomplishments of the project.

After realizing that running PQC algorithms on such lightweight systems was more challenging than expected, the focus shifted to implementation on the Raspberry Pi. Throughout the project, the benchmark evolved significantly while maintaining its primary goal: simulating a real-world scenario in which an embedded system securely transmits data to a server using post-quantum-safe cryptography. The Realization section provides an overview of the mid-level implementation details.

At its core, the project aimed to establish secure communication between a client and a server, as illustrated in Figure Figure 1:

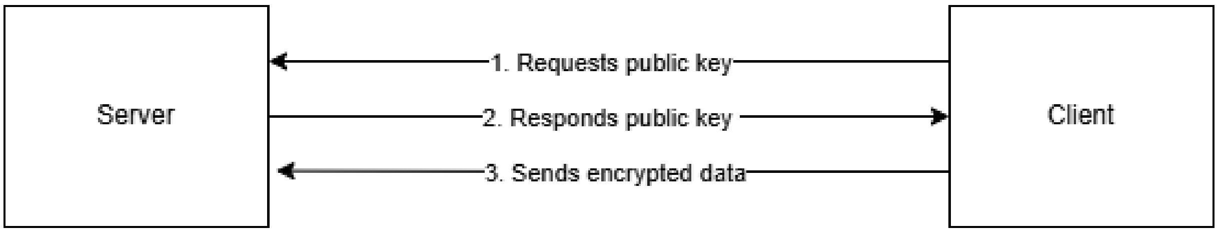

Figure 1: Communication between Server and Client

For the cryptographic algorithm implementations, the PQClean library was used. Additionally, for the Rust implementation, the Kyber library from Argyle-Software was utilized.

PQClean is a library that provides 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 to ensure security and auditability. The library includes implementations of NIST PQC standardization candidates while ensuring resistance to side-channel attacks. Written in standard C, PQClean is designed for portability and easy integration into security-critical applications, serving as a reliable 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.

2.2 Required Packages

The following Python packages were required for the implementation:

- pycryptodome
- flask
- requests

- paramiko
- pandas
- python-dotenv

2.3 Compiling C-Based PQClean Algorithms

To execute the PQClean library's C-based algorithms from Python, the ctypes module was used. The C source files needed to be compiled beforehand. To automate this process, Python scripts were written to generate a Makefile for each algorithm and execute make.

This process resulted in a build directory containing two subdirectories:

crypto_sign/ (for signature mechanisms) crypto_kem/ (for key encapsulation mechanisms) Since PQClean follows a strict guideline for function interfaces, calling the compiled implementations was straightforward.

2.3.1 Function Headers for Key Encapsulation Mechanism (KEM)

The function headers for KEM algorithms in PQClean always follow this format:

```
  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 example, the function headers for Kyber512 are:

```
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);
```

2.3.2 Function Headers for Digital Signature Mechanism

For post-quantum signatures, PQClean provides the following standard function headers:

```
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 Dilithium2, the corresponding function headers are:

```
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);
```

2.4 Rust Implementation

The Rust-based implementation follows a different interface. To accommodate this, a wrapper was created in the Rust directory. A shell script was also developed to compile this wrapper and generate the necessary binaries. These binaries are then moved into the *build/crypto_kem/* directory.

(See Further Notes for Rust installation details.)

2.5 First implementation

The first implementation differed significantly from the final result. The communication between the server and client in this initial version is shown in Figure Figure 2:

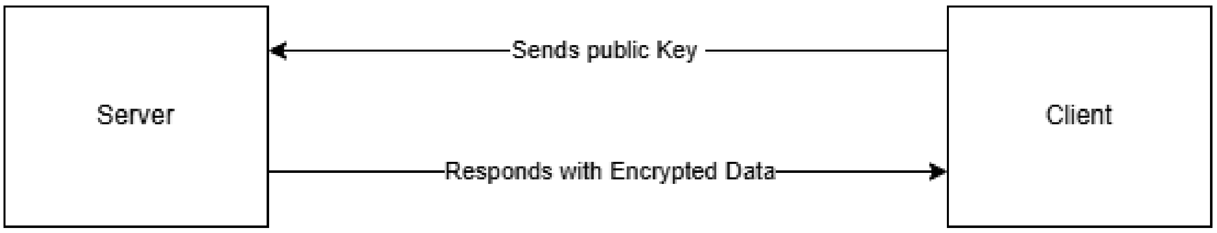

Figure 2: First Implementation

2.5.1 Initial Communication Flow

1. The server receives a public key for a key encapsulation mechanism (KEM) and returns encrypted data.
2. The server also signs the hash of the encrypted content.
3. The server benchmarks several operations, including:
   - Performing key encapsulation
   - Encrypting data
   - Hashing the encrypted data
   - Signing the hash
4. The benchmark results are sent back to the client.
5. The client verifies and decrypts the response while also measuring execution time.
6. All timing results are saved to a file.

2.5.2 Problem with the Initial Implementation

While this implementation successfully performed post-quantum encryption and benchmarking, the client-server roles were misaligned.

- In this setup, the server was responsible for measuring data (e.g., temperature) and waiting for the client to request it.
- This meant that each device would need to run a server, which is not ideal for real-world applications.

A more practical solution would be for the sensor device (measuring temperature) to act as the client, while the data-collecting device acts as the server.

2.6 Final Implementation

The final implementation follows the improved client-server model, as illustrated in Figure Figure 1:

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. 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:

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:

2.6.1 Communication Flow

1. The client requests a public key for a key encapsulation mechanism (KEM).
2. Using this public key, the client generates an AES key and encapsulates it.
3. The AES key is then used to encrypt the data.
4. The encrypted data is hashed and signed using a post-quantum signature mechanism.
5. Each process is timed, and the results are saved to a CSV file.
6. The client sends the following data to the server:
   - Encrypted data
   - Signature
   - Public signature key
   - Used signature and KEM algorithms
   - Encapsulated AES key
   - IV (Initialization Vector)
7. The server:
   - Hashes the encrypted data and verifies the signature.
   - Decapsulates the AES key and decrypts the data.
   - Measures execution time and logs it in a CSV file.

At startup, both the server and client generate and log the key pair generation times for each algorithm.

To run the benchmark, install the server on Device A and the client on Device B. Then:

- Start the server on Device A
- Start the client on Device B

The benchmark runs continuously until the client is stopped.

2.7 Controller

Since the server and client only generate their keys once at startup, comparing key generation times between algorithms requires restarting both processes.

2.7.1 Automating Benchmark Execution

- Manually restarting the server and client at intervals is not practical, especially on lightweight devices where key generation is time-consuming (see Evaluation).
- Instead, a Controller Script automates this process.

2.7.2 Controller Script Features

- Uses SSH and SCP to manage the benchmark and collect generated files.
- Configurable settings:
  - Hostnames
  - SSH key locations
  - Number of iterations
  - Time per iteration
- At the start of execution, the controller deletes all CSV files on the devices (⚠ caution advised).

Once the benchmark completes, the script downloads and consolidates all generated files into the following CSV files:

- client_timings.csv
- server_timings.csv
- key_generation_times.csv

3 Data

4 Further Notes

This section provides important details to ensure the benchmark runs correctly.

4.1 Environment Files (.env)

- Create .env files in the respective directories with the following format:

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

- The server and client retrieve their names from these files to correctly log data in the CSV files.

4.2 Controller Script and Installation Process

- The controller script includes a shell script that ensures all required dependencies are installed before running the benchmark.
- If any installation step fails, the benchmark will not start to prevent incomplete or inconsistent results.
- The controller script automatically performs the full installation process, including:
  - Compiling the necessary binaries from the PQClean library.

  - Building the Rust wrapper.

4.3 SSH Authentication

- The controller script authenticates via SSH keys.
- RSA keys seem to cause issues with authentication.
- Instead, generate an Ed25519 SSH key using the following command:

```
ssh-keygen -t ed25519
```

4.4 Rust installation

- Install only the minimal Rust toolchain on the Raspberry Pis, as the full toolchain is too large and unnecessary for this project.

5 Evaluation

6 Conclusion