Contents

1 Introduction

The primary objective of the project is to find a Post Quantum Cryptography (PQC) replacement for the conventional cryptography in the Password Authenticated Connection Establishment (PACE) protocol, with the development of a prototype being the secondary objective.

The group working on the project in the previous semester did lay a good foundation with a lot of research. A summary of their work:

- PACE and PQC:
  The projects Wiki[5] contains a lot of research on theoretical aspects of the project.
- PQC scheme to replace the Diffie-Hellman scheme:
  Multiple NIST candidates were evaluated and Kyber was chosen for this project, which turned out to be a good choice, as it became a standardization candidate „“[7, p. 29][1]
- Hardware for prototype:
  Different hardware has been compared and development boards were purchased from STMicroelectronics.
- Implementation:
  Some efforts were made to have PACE and Kyber combined. Code was running in a Virtual Machine and partly working on the development board. However an important property of the PACE protocol (a random nonce that is encrypted with the PIN should be used for running the protocol but the final key must not be dependent on the nonce) was still unfulfilled.

However, there was still a lot of work left to do, especially on the development side of the project. The main goal for summer semester 2022 was to replace the protocol with unsafe key generation and develop a working prototype which runs on a development board. d

2 Theory

2.1 Basic cryptography

Cryptography is the study of secure communication techniques. A large part of that study is encryption and decryption, which uses mathematical

[1] https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms

methods to transform a plaintext into a ciphertext or vice versa, to ensure confidentiality. This is achieved by using one of various encryption algorithms and other cryptographical functions such as secure hash functions. The field can be divided in three caterories:

- symmetrical cryptography
- asymmetrical cryptography
- cryptographic hash function

2.1.1 Symmetrical cryptography

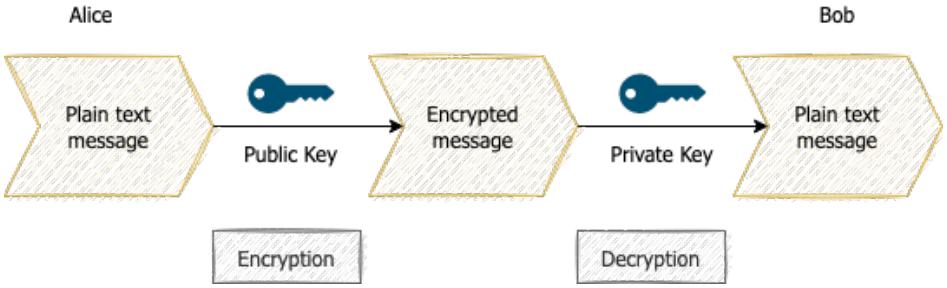
This is the simplest type of encryption which involves only one key to encrypt and decrypt the information. The sender and the recipient need to know the secret key in order to be able to encrypt their own messages and to decrypt the other side's message. Examples: AES, DES, RC6. The advantage of symmetrical cryptography is good performance, but on the other side both parties should have exchanged the key before the communication.

Advanced Encryption Standard (AES) is the most widely used symmetric cryptographic method and was standardized by NIST in October 2020 to succeed the less secure DES algorithm. Its key size can be 128-, 192- or 256-bit long and its block size is fixed at 128-bit.

2.1.2 Asymmetrical cryptography

Asymmetrical cryptography is known as public-key-cryptography.

Public-Key Encryption (PKE) is a method to encrypt and decrypt data using a public and a private key. The public key has the function to encrypt the data and, as the name indicates, the public key of one party is available to everybody. After the encrypted data is sent to the recipient, the recipient decrypts the message using the private key. The encryption of any message can be only done with the public key and the decryption can be only be performed with the corresponding private key.

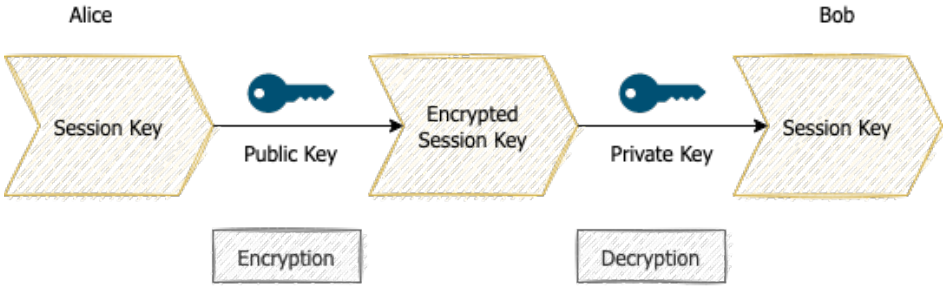


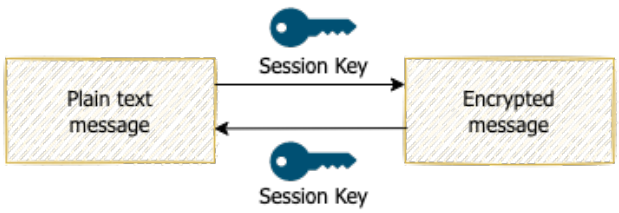
Summarizing, the Public Key Encryption consists of three algorithms:

1. Key generation algorithm which generates public and private key pair.
2. Encryption algorithm which takes a plain text and public key and outputs a ciphertext.
3. Decryption algorithm which takes a cipher text and private key to output original message.

Public Key Encryption is known as asymmetric encryption and is widely used, for example in TLS.

Key Encapsulation Mechanism (KEM) is a method for secure transmission of session keys between two parties using public key encryption. Due to the fact that using PKE for encryption of longer message is tedious, key encapsulation mechanisms are used to encrypt the symmetric session key with the public key. Like in PKE, the encrypted session key can be decrypted with the associated private key.



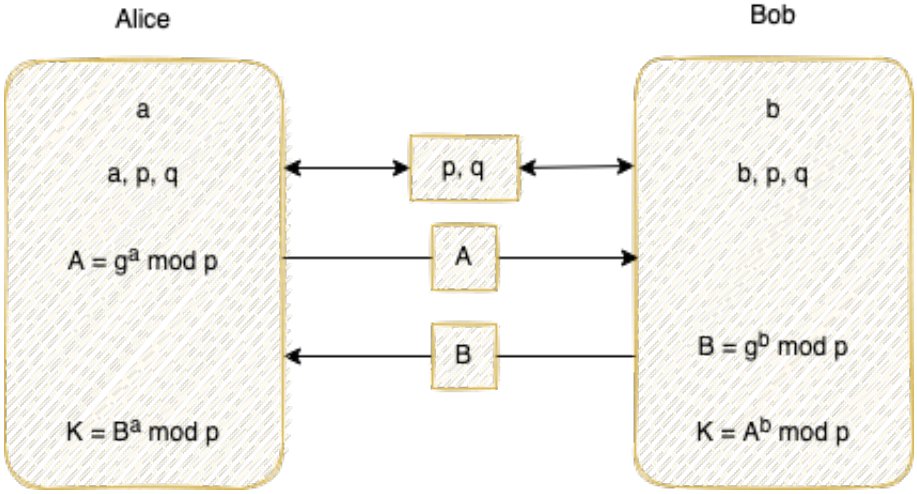


After the session key is known to both parties, Alice and Bob can use the session key to encrypt their outgoing messages and decrypt the incoming message from each other. Like PKE, it consists of three algorithms:

1. Key generation algorithm which generates public and private key pair.
2. Encryption algorithm which takes a plain text and public key and outputs a ciphertext and a session key.
3. Decryption algorithm takes a cipher text and private key to compute a session key. Because of the fact that asymmetric keys are short, KEM uses an algorithm to generate a random element in the finite group underlying the public key system and derives the symmetric key by hashing its element, so that padding is not necessary.

2.1.3 Popular Cryptographic Schemes

Diffie-Hellman Key Exchange Procotol is a method, which allows two parties to establish the common session key over an insecure channel, so that a shared secret is know for both parties. The knowledge of the shared secret is necessary to calculate the common session key.



1. Alice and Bob create own private values: a (Alice) and b (Bob).
2. Alice and Bob agree on common values and exchange them: p, q.
3. Alice computes A and sends it to Bob.
4. As return, Bob computes B and sends it to Alice.
5. Alice is able to calculate the session key K.

6. Bob calculates also the session key K.

7. Now they are able to use the session key K to encrypt and decrypt messages.

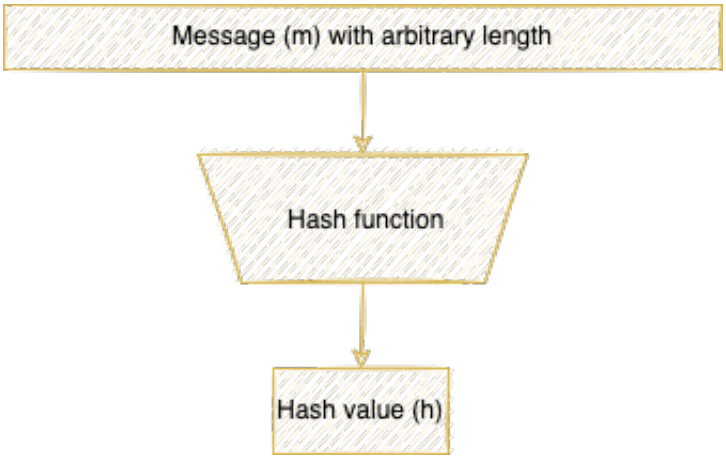
Using this method both parties are able to calculate the session key without having the knowledge of the second party's private value. The session key is established on the basis of the private key of both parties. The Diffie-Hellmann Procotol can be also used for more than two parties.

RSA (Rivest–Shamir–Adleman) is a one example of Public-Key Encryption mentioned before and is named after their inventors. A RSA cryptosystem contains public key and private key and the core of the key generation are prime numbers. They are kept secret, so that the encrypted message can be only decrypted by the party knowing the prime numbers. The security of this cryptosystem is based on the difficulty of factoring prime numbers, which is called the factoring problem in mathematics.

Elliptic Curve Cryptography (ECC) is another cryptographic technique, much more powerful than RSA and considered to be the next generation implementation of public key encryption. ECC uses public-key approach based on how elliptic curves are structured algebraically over finite fields to encrypt and decrypt the messages. The relevant differences between RSA and ECC consists in smaller key sizes than in RSA and better security: RSA creates public-private key pair based on prime number factorisation, while ECC creates key pair more mathematically difficult to break. To compare, a 2048-bit length RSA key achieves the same level security as a 224-bit length ECC key.

2.1.4 Hash function cryptography

Hash functions are deterministic functions used to map arbitrary length input data to a fixed n-length output (hash value).



The common hash functions generate values between 160 and 512 bits, the most popular are MD and SHA family. The hash function should possess the following properties:

- **collision resistance** - two different inputs of any length should not give the same output
- **pre-image resistance** - it should be mathematically hard to reverse the hash function
- **second pre-image resistance** - if we know the input and its hash, it should be hard to find the different input value with the same hash

Generally, the efficiency of hash calculation is considered as fast operation and as result they are much faster than symmetric cryptography. Based on their properties, they are used for password storage and for data check integrity.

SHA (Secure Hash Algorithm) The SHA hash functions family consists of four SHA-algorithms: SHA, SHA1, SHA2 and SHA3. Although they are considered to be in the same family, they differs strong in their structure. SHA-3 algorithm was released in 2012 as NIST standard for hash functions.

2.2 Post-Quantum Cryptography

Post-Quantum Cryptography refers to the part of cryptography, that concerns itself with the security of cryptographic methods against quantum computers. The current cryptographic algorithms like RSA and ECC are based on mathematical problems like the factorization problem, the discrete logarithm problem and the elliptic curve discrete logarithm problem. All of

them can be easily broken by a powerful quantum computer. In order to solve this problem, new types of algorithms are needed.

Types of PQC algorithms:

- Lattice-based cryptography
- Multivariante cryptography
- Hash-based cryptography
- Supersingular elliptic curve isogeny cryptography
- Code based cryptography

2.2.1 Lattice-based cryptography

A lattice is a mathematical structure representing an infinite grid of points. To build the lattice a set of basis vectors is needed. In this kind of cryptography the complexity increases with dimensions and amount of basis vectors. Lattice based cryptography can be based on different problems (there are more):

- Closest vector: given a lattice and a point - find the closest vector to point
- Shortest vector: given a lattice and a point - find the shortest non zero vector

NIST Candidates that use lattice based algorithms:

- NTRU
- CRYSTALS-KYBER
- SABER
- CRYSTALS-Dilithium
- FALCON

Learning with Errors (LWE): A random matrix A, a secret matrix s and an error matrix e are choosen. Error matrix e acts as noise and is needed because otherwise gaussian elimination could be used to easily compute s. A acts as the public key and s and e as the private key. Keys are exchanged using Diffie Hellman (computing mixed key by A*s+e). For Ring LWE the matrix is exchanged for a polynomial ring.

Lattice scheme types are described by their underlying lattice and if they can be structured or unstructured. Lattices based on some kind of learning with errors or rounding will require some sort of error correction.

Unstructured lattices have increased security but will have larger key sizes and slower performance (because random data has to be generated and more operations are required for public key and ciphertext computation). Examples:

- Frodo
- Round5 (can be implemented structured or unstructured)

Structured Lattices This kind of lattices have an underlying algebraic structure that might be more predictable than unstructured lattices. But therefore overall performance is better.

- Saber
- Kyber
- NewHope
- LAC

Ideal lattices are a subgroup of structured lattices and are the basis for RLWE lattices. They drastically decrease the amount of parameters that are required to describe the lattice. Examples:

- NTRU KEM

2.2.2 NIST Post-Quantum Cryptography Standardization Process

The National Institure for Standards and Technology started a process to find suitable candidates for PQC standardization. Because winners will probably become the standard, which will influence the industry, it made sense to decide on a scheme that is part of this process. Decisions for which scheme to use were made in the first semester of this MPSE before round three ended. A note on NISTs decisions can be read further down below.

NIST Round 2 KEM candidates All LWE-R based PKE/KEM NIST Candidates contain schemes based on the LPREncryption framework. Schemes can differ in parameters, underlying ring, relative sizes of rounding moduli, Error distribution and choice of Error Correcting Code. Most of them use either gaussian sampling or centered binomial sampling for creating the error vectors.

- **Frodo**: standard LWE, power of 2 modulus (max $2\hat{1}6$) and rounded gaussian error distribution KEM
- **NewHope**: based on RLWE, power of 2 cyclomatic ring, parameters chosen to fit NTT based polynomial multiplication, secrets and errors sampled from CBD KEM IDEAL Lattice.
- **Kyber**: Based on MLWE, ring is based on ring used in NewHope, CBD for sampling, ciphertexts are compressed, disadvantage: pub module a with $k\hat{2}$ polynomials (k times more compared to RLWE with similar security) KEM.
- **Saber**: Similar to Kyber based on MLWR, same ring as same ring Kyber, power of 2 modulus (helps with simplifying modular arithmetics and accelerates sampling operations) KEM.
- **LAC**: RLWE, byte level moduli (good for bandwidth efficiency), similar cyclomatic ring of NEWHope and sample secrets and errors from CBD - high decryption failures corrected using BCH and D2 Codes to make error rate negligible KEM, KEX; PKE.
- **Round5**: IND-CPA secure KEMs and IND-CCA secure PKAs, based on GLWR using sparse ternary secrets (small number of nonzero coefficient 3er vectors) - use LWR/RLWR/MLWR problem, very flexible, Operating ring is prime cyclomatic ring, modulus is power of $2(< 2^{16})$ lightweight error correcting code called XEF.
- **ThreeBears**: Based on I-MLWE, based on reconciliation based Noisy DH that is converted into KEM, Involves computation with big integers and simple BCH ECC KEMs.
- **NTRU based KEMs** won't require Error Correction! have compact keys and ciphertexts.
- **NTRUKEM**: merges NTRUEncrypt and NTRUHRSS variants of NTRU-SXY KEM and support parameter sets of them. KEMs.

- **NTRUPrimeKEM**: Can use both a public key similar to LWR that is rounded and a public key in quotient form.

NIST Update on closing round 3 On July 5th 2022 NIST announced their decisions regarding completing round 3. Four candidates were selected for standardization and four more are selected for an additional fourth round. Out of the four candidates three are digital signatures (Crystals-Dilithium, Falcon and SPHINCS+). The only selected Public-Key Encryption scheme is CRYSTALS-Kyber.

On upcoming process:

"NIST will create new draft standards for the algorithms to be standardized and will coordinate with the submission teams to ensure that the standards comply with the specifications. As part of the drafting process, NIST will seek input on specific parameter sets to include, particularly for security category 1. When finished, the standards will be posted for public comment. After the close of the comment period, NIST will revise the draft standards as appropriate based on the feedback received. A final review, approval, and promulgation process will then follow".[2]

Candidates in the fourth round are all Public-Key Encryption/KEMs. Those candidates are BIKE, Classic McEliece, HQC and SIKE, and their respective teams have been given another chance to update their specifications until October 1st.

2.2.3 Other PQCs that might not fulfill our needs

Hash-based cryptography

- Based on hash functions and combine a one-time signature scheme with a Merkle tree structure.

Code-based cryptography

- Based on error correcting codes.
- NIST Candidates that use code based algorithms: Classic McEliece (3. round finalist), BIKE and HQC

Supersingular elliptic curve isogeny cryptography

- Based on supesingular elliptic curves and graphs and offers forward secrecy.

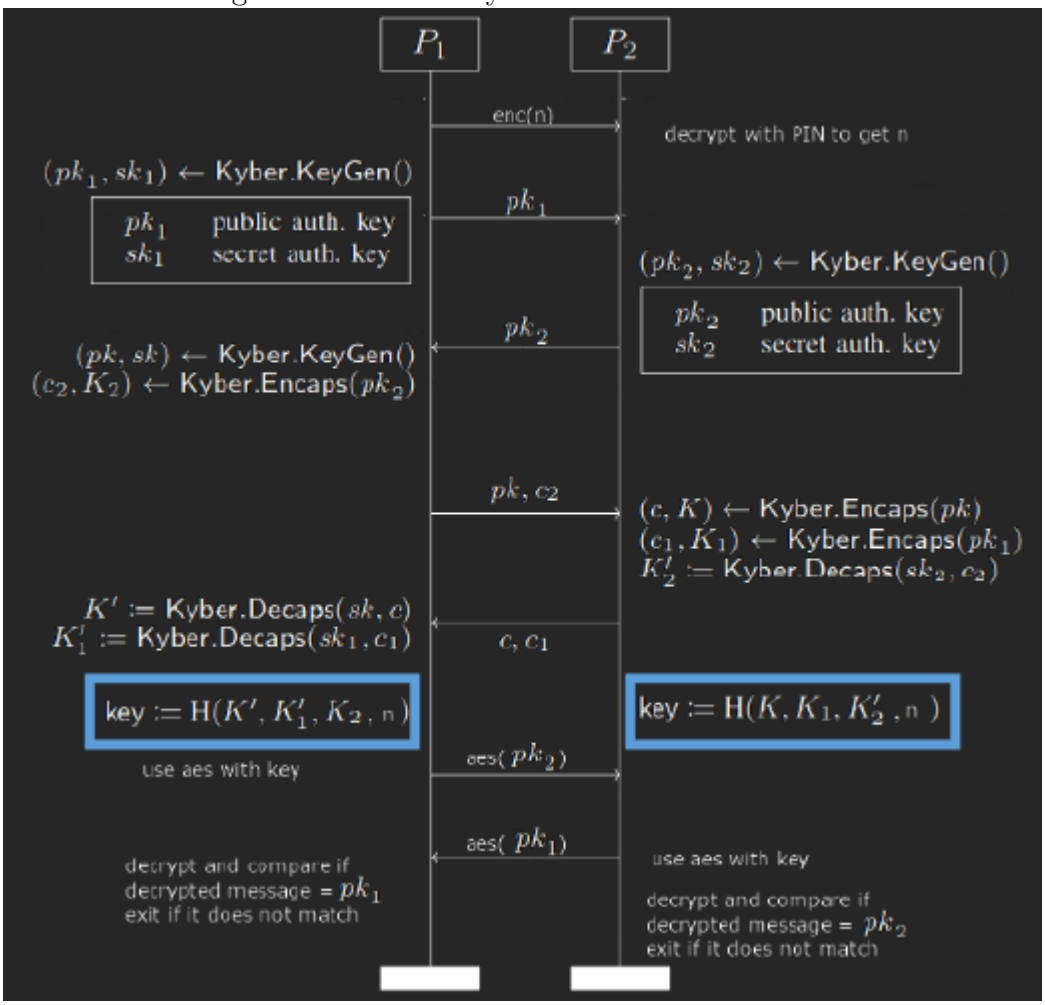
Multivariate-based cryptography

- Based on mutlivariate equations.
- Used by Rainbow scheme.

2.3 Previous state of the protocol

In the previous semester, the group added the nonce to the keying material as a temporary measure, but this solution is insecure as with brute forcing the nonce (which would take 10^4 guesses at worst) the key can be recovered.

Figure 1: Modifed Kyber PAKE from WS2021



2.4 Current state of the protocol

Ding et al. proposed a PAKE based on RLWE [3] which serves alongside Kyber KEM as the base of the newly constructed Kyber-Ding PACE. An overview is shown in Figure 1. The Protocol has similar steps to the PACE protocol. First a nonce is randomly drawn and encrypted using the hashed PIN, by the eID Card. Additionally the lattice base A is generated and gets send alongside the encrypted nonce to the terminal. In the *mapping phase* both terminal and card compute their public keys using *Learning With Errors (LWE)*. Terminal sends its regular public key while the card adds the hashed nonce to its public key before sending. This ensures that the correct key can only be achieved if the terminal is able to recover the right public key of the card, which is only possible when the terminal is able to decrypt the nonce using the right PIN. After the mapping phase the *two-path pre Keys phase* starts. Both Card and Terminal generate a message each, which is then encapsulated with the others public key. The encapsulated message is then exchanged and decapsulated. The Recovered message is then encapsulated again to check if decapsulation was successful and gain the pre key. The pre key is then used in the *Authentication phase* to derive an encryption and two message authentication codes. One of the macs, the lattice base and each others public key are packed into an authentication token to ensure both parties were able to generate the right key. The other mac Key and the encryption key are then used to establish a session and start encrypted communication.

2.5 Further Theory efforts

Besides getting to know and fully understanding the new protocol in its entirety we created a run of the protocol on paper, with the help of a python script. This was done to help model attacks in the future as the regular Kyber instantiations use quite large numbers and long messages which are hard to overlook for the human observer. The calculations can be found in annex A. Furthermore we looked into finding attacks on Kyber-Ding PACE and security proofs using security games in general.

2.5.1 Modified Kyber-Ding-PACE

Alice		**Bob**
password π		password π
	Exchange nonce	
$K_\pi = \mathcal{H}(\pi\|\|0)$ choose $n \leftarrow \mathbb{Z}_q$ $z = \mathcal{C}(K_\pi, n)$ generate $\mathbf{A} \in \mathbb{R}^{k\times k}_{q=3329}$		$K_\pi = \mathcal{H}(\pi\|\|0)$
	$\xrightarrow{\mathbf{A},z}$	
		$n = C^{-1}(K_\pi, z)$
	Mapping	
generate $\mathbf{s}_a, \mathbf{e}_a \in \mathbb{R}^k_{\eta=3}$ $\mathbf{t}_a = \mathbf{A}\mathbf{s}_a + \mathbf{e}_a$		generate $\mathbf{s}_b, \mathbf{e}_b \in \mathbb{R}^k_{\eta=3}$ $\mathbf{t}_b = \mathbf{A}\mathbf{s}_b + \mathbf{e}_b$
	$\xleftarrow{\mathbf{t}_b}$	
	with hash $\mathbf{H}$: $\{0,1\}^{256} \to \mathbb{R}^k_{q=3329}$	
$\mathbf{p}_a = \mathbf{t}_a + \mathbf{H}(n)$	$\xrightarrow{\mathbf{p}_a}$	$\mathbf{t}_a = \mathbf{p}_a - \mathbf{H}(n)$
	two-path preKeys	
generate $m_a \leftarrow \{0,1\}^{256}$ $(\hat{K}_a, (\mathbf{r}_a, \mathbf{e}_{a1}, e_{a2})) = \mathbf{G}(\mathbf{H}(\mathbf{t}_b), m_a)$		generate $m_b \leftarrow \{0,1\}^{256}$ $(\hat{K}_b, (\mathbf{r}_b, \mathbf{e}_{b1}, e_{b2})) = \mathbf{G}(\mathbf{H}(\mathbf{t}_a), m_b)$
	with hash $\mathbf{G}$: $\mathbb{R}^k_{q=3329}, \{0,1\}^{256} \to$ $\{0,1\}^{256}, (\mathbb{R}^k_{\eta=3}, \mathbb{R}^k_{\eta=3}, \mathbb{R}^k_{\eta=3})$	
$c_a = \begin{cases} \mathbf{u}_a = (\mathbf{A}^T\mathbf{r}_a + \mathbf{e}_{a1}) \\ v_a = \mathbf{t}_b^T\mathbf{r}_a + e_{a2} + \lceil \frac{q}{2} \rfloor \cdot m_a \end{cases}$		$c_b = \begin{cases} \mathbf{u}_b = (\mathbf{A}^T\mathbf{r}_b + \mathbf{e}_{b1}) \\ v_b = \mathbf{t}_a^T\mathbf{r}_b + e_{b2} + \lceil \frac{q}{2} \rfloor \cdot m_b \end{cases}$
	$\xrightarrow{c_a=(\mathbf{u}_a,v_a)}$ $\xleftarrow{c_b=(\mathbf{u}_b,v_b)}$	
$m_b^* = (v_b - \mathbf{s}_a^T\mathbf{u}_b)$ $\hat{K}_b^*, (\mathbf{r}_b^*, \mathbf{e}_{b1}^*, e_{b2}^*)) = \mathbf{G}(\mathbf{H}(\mathbf{t}_a), m_b^*)$ $c_b^* = \begin{cases} \mathbf{u}_b^* = (\mathbf{A}^T\mathbf{r}_b^* + \mathbf{e}_{b1}^*) \\ v_b^* = \mathbf{t}_a^T\mathbf{r}_b^* + e_{b2}^* + \lceil \frac{q}{2} \rfloor \cdot m_b^* \end{cases}$ generate $z_a \leftarrow \{0,1\}^{256}$ $\hat{K}_b = \begin{cases} \hat{K}_b^* \text{ if } c_b = c_b^* \\ z_a \text{ if } c_b \neq c_b^* \end{cases}$		$m_a^* = (v_a - \mathbf{s}_b^T\mathbf{u}_a)$ $\hat{K}_a^*, (\mathbf{r}_a^*, \mathbf{e}_{a1}^*, e_{a2}^*)) = \mathbf{G}(\mathbf{H}(\mathbf{t}_b), m_a^*)$ $c_a^* = \begin{cases} \mathbf{u}_a^* = (\mathbf{A}^T\mathbf{r}_a^* + \mathbf{e}_{a1}^*) \\ v_a^* = \mathbf{t}_b^T\mathbf{r}_a^* + e_{a2}^* + \lceil \frac{q}{2} \rfloor \cdot m_a^* \end{cases}$ generate $z_b \leftarrow \{0,1\}^{256}$ $\hat{K}_a = \begin{cases} \hat{K}_a^* \text{ if } c_a = c_a^* \\ z_b \text{ if } c_a \neq c_a^* \end{cases}$
	Authentication	
$K = \mathbf{KDF}(\hat{K}_a, \hat{K}_b)$ $K_{enc} = \mathcal{H}(K\|\|1)$ $K_{mac} = \mathcal{H}(K\|\|2)$ $K'_{mac} = \mathcal{H}(K\|\|3)$ $T_A \leftarrow \mathcal{M}(K'_{mac}, (\mathbf{t}_b, \mathbf{A}))$		$K = \mathbf{KDF}(\hat{K}_a, \hat{K}_b)$ $K_{enc} = \mathcal{H}(K\|\|1)$ $K_{mac} = \mathcal{H}(K\|\|2)$ $K'_{mac} = \mathcal{H}(K\|\|3)$ $T_B \leftarrow \mathcal{M}(K'_{mac}, (\mathbf{t}_a, \mathbf{A}))$
	$\xrightarrow{T_A}$ $\xleftarrow{T_B}$	
abort if T_B invalid		abort if T_A invalid
	Establish Session	
key $= (K_{enc}, K_{mac})$ sid $= (\mathbf{t}_a, \mathbf{t}_b, \mathbf{A})$ pid $= \epsilon$		key $= (K_{enc}, K_{mac})$ sid $= (\mathbf{t}_a, \mathbf{t}_b, \mathbf{A})$ pid $= \epsilon$

3 Implementation

3.1 Technology Stack

The old Technology Stack, consisting of *C, Kyber, openPACE, OpenSSL and Docker*, has been simplified, as it introduced a lot of overhead.

OpenPACE was previously used for some cryptographic primitives like hash functions, which have been replaced by functions implemented by Kyber. OpenSSL was introduced as a dependency by OpenPACE, which was then replaced by WolfSSL due to the size of OpenSSL. This, however, led to build failures due to compatibility issues. Since OpenPACE was removed, OpenSSL/WolfSSL have been removed too.

The application does not need to be scalable, since only one client and server are interacting at any given time. And since a Virtual Machine is used for Development, Docker was deemed unnecessary and thus removed.

Additionally, it was decided to develop the application in C++ instead of C, to make development easier by having access to features such as classes. This adding some additional memory requirements is not an issue, as the removal of OpenPACE and OpenSSL greatly reduced the memory footprint of the application.

As a result of these changes, the current Technology Stack consists of C++ and the Kyber library.

3.2 Changes to Kyber

The following modifications of Kyber were made based on the Ding Kyber PACE (see here) in the following classes:

- indcpa.cpp, function indcpa_keypair
- kem.cpp, function crypto_kem_keypair

In the code two additional functions were created and above functions were used as a basis. Both functions were modified and named after original functions and marked with ding. The functions are called in AsymmetricKey.cpp.

3.2.1 crypto_kem_keypair_ding

This function is used to generate public and private key for CCA-secure Kyber key encapsulation mechanism. The function was modified, so that the public seed is used for the generation of the key pair.

```
int crypto_kem_keypair_ding(uint8_t *pk,
                            uint8_t *sk,
                            uint8_t *pubseed)
{
    size_t i;
    indcpa_keypair_ding(pk, sk, pubseed);
    for(i=0;i<KYBER_INDCPA_PUBLICKEYBYTES;i++)
        sk[i+KYBER_INDCPA_SECRETKEYBYTES] = pk[i];
    hash_h(sk+KYBER_SECRETKEYBYTES-2*KYBER_SYMBYTES,
           pk, KYBER_PUBLICKEYBYTES);
    /* Value z for pseudo-random output on reject */
    randombytes(sk+KYBER_SECRETKEYBYTES-KYBER_SYMBYTES,
           KYBER_SYMBYTES);
    return 0;
}
```

The following modifications were made in the function:

- parameter pubseed was added to the function header.
- the function indcpa_keypair_ding was used instead of indcpa_keypair. To integrate the public seed, a modification in the called function indcpa_keypair need to be done.

3.2.2 indcpa_keypair_ding

```
void indcpa_keypair_ding(uint8_t pk[KYBER_INDCPA_PUBLICKEYBYTES],
                         uint8_t sk[KYBER_INDCPA_SECRETKEYBYTES],
                         uint8_t pub_seed[2*KYBER_SYMBYTES])
{
    unsigned int i;
    uint8_t *publicseed = pub_seed;
    uint8_t *noiseseed = pub_seed + KYBER_SYMBYTES;
    uint8_t nonce = 0;
    polyvec a[KYBER_K], e, pkpv, skpv;

    randombytes(noiseseed, KYBER_SYMBYTES);
    hash_h(noiseseed, noiseseed, KYBER_SYMBYTES);

    gen_a(a, publicseed);

    for(i=0;i<KYBER_K;i++)
        poly_getnoise_eta1(&skpv.vec[i], noiseseed, nonce++);
    for(i=0;i<KYBER_K;i++)
        poly_getnoise_eta1(&e.vec[i], noiseseed, nonce++);

    polyvec_ntt(&skpv);
```

```
    polyvec_ntt(&e);

    // matrix-vector multiplication
    for(i=0;i<KYBER_K;i++) {
        polyvec_basemul_acc_montgomery(&pkpv.vec[i], &a[i], &skpv);
        poly_tomont(&pkpv.vec[i]);
    }

    polyvec_add(&pkpv, &pkpv, &e);
    polyvec_reduce(&pkpv);

    pack_sk(sk, &skpv);
    pack_pk(pk, &pkpv, publicseed);
}
```

The following modifications were made in the function:

- the header of the functions was extended by public seed.
- buf variable was deleted from the code.
- *publicseed and *noiseseed are not defined as const anymore
- instead of using the variable buf, pub_seed given as a parameter from the function call (see the first modification) was used. before the modification the noiseseed, which is used for the error generation, was dependent of the publicseed. An 512 bit hash was created from 256 bit publicseed and the second half of the hash was used as noiseseed. The dependency was removed, so the communication partners wouldn't generate the same keys.

3.3 Current state

In its current state, the application implements the complete flow of our modified Kyber-Ding-PACE protocol, starting with the input of the PIN and ending with the generation of the session properties.

Secure exchange of messages afterwards using AES-CBC is also working.

3.3.1 Development environment

Depending on your choice between using the preconfigured virtual machine or creating a new one, see the following steps for each possibility.

3.3.2 Case 1: Preconfigured virtual machine - Install the project

Downloads

- Virtual Box 6.1
- preconfigured Linux Mint - ask Nouri Alnahawi for download link since it's too big for the repository

Steps

1. Install Virtual Box
2. Click on File - Import appliance...
3. Choose the downloaded VM and import it into the Virtual Box
4. Now you can log in using the following credentials: User: linuxmint Initial password for VM's: linuxmint
5. Start CLion in the virtual machine in Terminal with
   cd /Downloads/clion-2022.1.3/bin
   ./clion.sh
6. Go to Help - Register and activate the Jetbrains license using your university credentials
7. In CLion terminal pull the newest version of the project:
   git pull

3.3.3 Case 2: New virtual machine - Configuring the virtual environment

Downloads

- CLion 2022.1.3 (Build 221.5921.27) for Linux
- Google Test

Steps

1. Before installing any further tools make sure to update the operating system using following commands:
   sudo apt update
   sudo apt upgrade
   and restart the system to confirm the changes.

2. Download and extract CLion inside the virtual machine.

3. Start CLion using the command ./clion.sh in the bin folder, where CLion is extracted.

4. Activate CLion using your educational license.

5. Install following tools in the terminal:
   sudo apt-get install git
   sudo apt-get install g++
   sudo apt-get install cppcheck

6. Install Google Test from Git: git clone https://github.com/google/googletest.git

Clone and configure the project repository

1. The first step is to clone the project: git clone https://code.fbi.h-da.de/aw/prj/athenepqc/mpse-eid-implementation.git using your university credentials.

2. Open the cloned project in CLion and click on Trust project

3. After the project is imported, click on Select CMakeLists.txt showed up in the top of the IDE and select CMakeList.txt from the Source folder.

4. To run the project, navigate to /Sources/client and click on Run - Edit configuration. Click on Add new... and choose CMake Application, click on Apply and OK

5. Go to Settings - Build, Execution, Deployment - Toolchains and be sure, that the field C++ Compiler has the value Detected: c++

6. Now you can run /Sources/client/client.cpp and /Sources/server/server.cpp using Run button in CLion.

7. Now the client connects to the server. The server asks for a PIN input. Currently the right PIN is "123457". The PIN can be changed in client.cpp. Enter the right or the wrong PIN for testing.

Integrating Google Test In case you are going to write some unit tests in the project, there are some test files already in /Sources/tests/ folder. To integrate them in the project, the following steps are necessary:

1. Install GoogleTest using Git: git clone https://github.com/google/googletest.git and extract its content into mpse-eid-implementation/Sources/tests
2. Add the line add_subdirectory(Sources/tests/googletest) in the CMakeLists.txt in tests folder.
3. If you are going to create some new unit tests, see the documentation of Google Test (for creating new tests it is necessary to edit the CMakeLists.txt file in test folder).

3.3.4 Experimental setup on the board

Requirements

- Installed and configured project for development on own computer (master branch)
- Installed STM32CubeIDE locally
- Board STM32L4R5ZI
- Cables: 2x MicroUSB / USB-C (depending on used converter)
- UART Converter

Downloads

- STM32CubeIDE for Linux

Setup

This documentation describes how to set up the experimental environment to bring the code on the board. It does not contain the information how to use the board in the productive environment. In our experiment we run the server on the own computer and the client on the STM32-board.

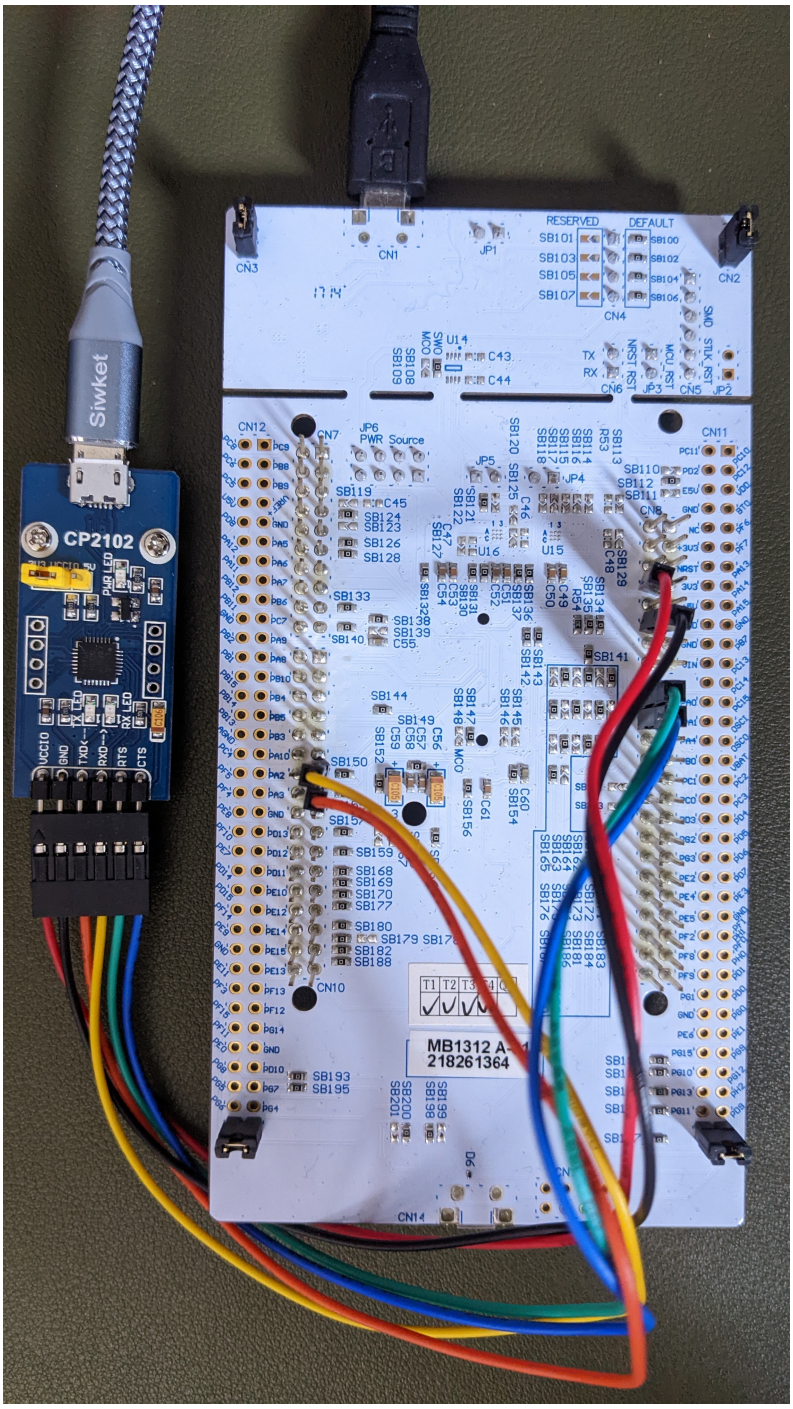


Figure 2: Experimental setup of the board.

Connect the board

1. Connect the Universal synchronous/asynchronous receiver/transmitter (USART) converter with the board as on the Figure 2.

    - **orange cable** - connection betweeen transmit data (TXD) on the USART converter and recieve data (RXD) on the board. This connection allows the board to receive the data.
    - **yellow cable** - connection between RXD on the USART converter and RXD on the board. This connection allows the board to receive the data.
    - **red cable** - power supply, need to be connected with 3V3 (3.3 V) or 5V. In our experiment we used the 3V3 connection.
    - **black cable** - need to be connected with ground (GND) on the board.
    - **blue cable** - connection between clear to send (CTS) on the converter and request to send (RTS) on the board. Flow control is highly recommended[4, p. 96]. Currently not used.
    - **green cable** - connection between RTS on the converter and CTS on the board.

2. Connect both universal serial bus (USB) cables with the computer:

    - **black cable** - connection between the board and the computer. This cable is used to transmit the code on the board.
    - **gray cable** - connection between the USART converter and the computer. This cable is used for the communication.

Run the experiment

1. Start STMCubeIDE on the computer, open the project as described in [1] and run it. In case this does not work, create a new project and copy the source code files into the respective folder. Please reload the project in order to enable the settings as defined in our code. Your user needs sufficient permissions to write to tty directly. In Ubuntu/Debian distributions this can be achieved by running as root (not recommended) or adding the user to the dialout group[8]. On other distributions this group might have a different name. Arch-Linux based distributions for example use the uucp group[9].

2. Start the server (terminal) on the computer.

3. Reset the board using the reset button, so that the program (client) runs from the beginning.

4. On the console you can see the data exchange between the client and the server. Additionally, the board uses the user LED's as status indicator. The LED's encode the board status as follows:

   - **fast blinking red LED** - receiving incoming data size,
   - **slow blinking red LED** - receiving actual data,
   - **fast blinking blue LED** - sending outgoing data size,
   - **slow blinking blue LED** - sending actual data.

4 Future work

During the project in the summer semester 2022 many tasks were completed, but there is still a lot to do in the future:

- Formal proof of security of the modified Kyber-Ding-Pace protocol: While research concerning the security of the protocol has been done, a formal proof is missing.

- Take a look at the implementation of Kyber in pqm4 [6]: The current project uses the reference implementation of Kyber. pqm4 contains different optimized implementations that could benefit the project.

- Fix the error causing the communication being one sided.

References

[1] [Online; accessed 28. Jul. 2022]. July 2022. URL: https://www.st.com/resource/en/user_manual/dm00598966-stm32cubeide-quick-start-guide-stmicroelectronics.pdf.

[2] *Announcing PQC Candidates to be Standardized, Plus Fourth Round Candidates | CSRC*. [Online; accessed 7. Aug. 2022]. July 2022. URL: https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4.

[3] Jintai Ding et al. "Provably secure password authenticated key exchange based on RLWE for the post-quantum world". In: *Cryptographers' Track at the RSA conference*. Springer. 2017, pp. 183–204.

[4] Warren W. Gay. *Mastering the Raspberry Pi*. New York, NY, USA: Apress. ISBN: 978-1-4842-0181-7. URL: https://link.springer.com/book/10.1007/978-1-4842-0181-7.

[5] *Home · Wiki · AW / PRJ / AthenePQC / MPSE eID Implementation · GitLab*. [Online; accessed 18. Jul. 2022]. July 2022. URL: https://code.fbi.h-da.de/aw/prj/athenepqc/mpse-eid-implementation/-/wikis/home.

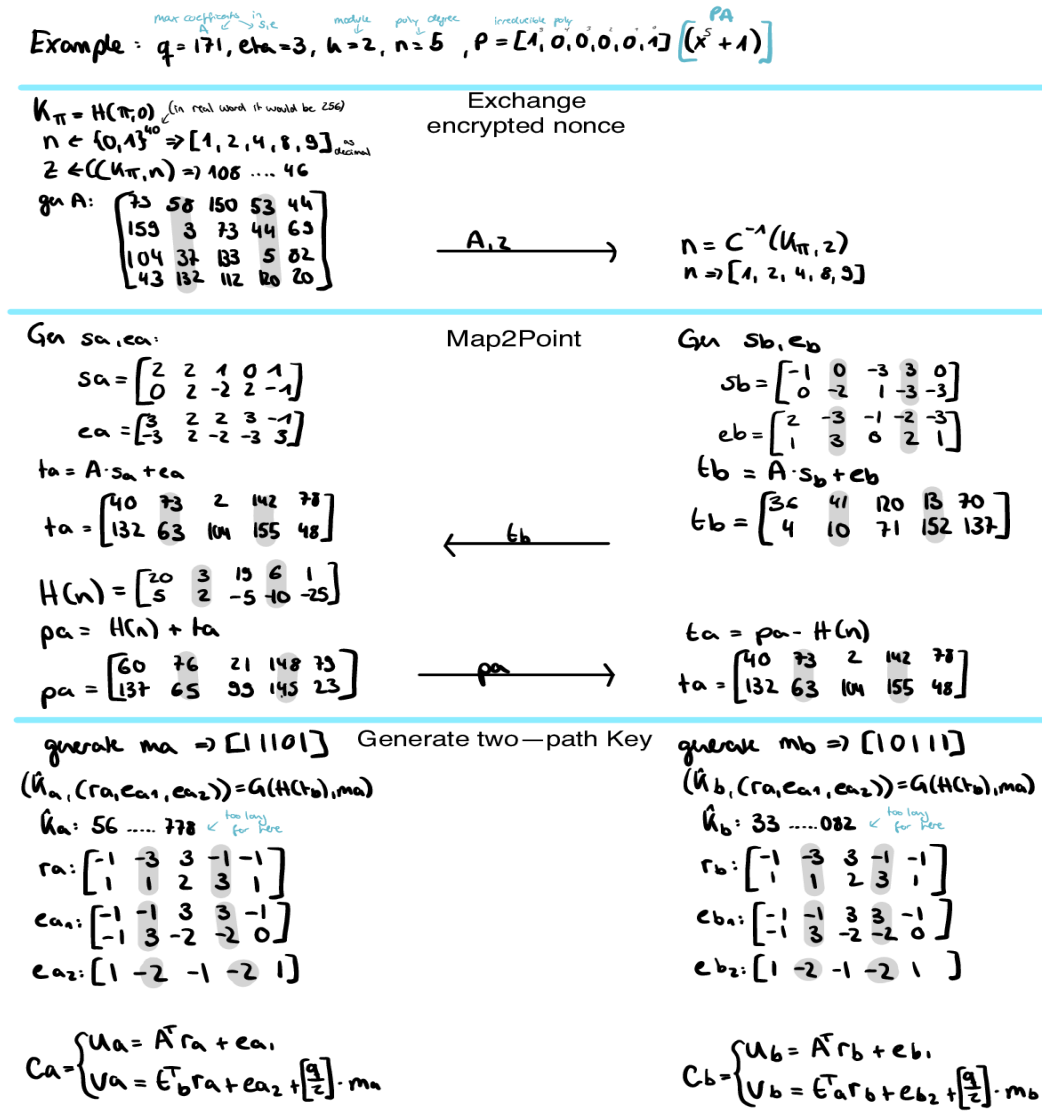
[6] Matthias J. Kannwischer et al. *PQM4: Post-quantum crypto library for the ARM Cortex-M4*. https://github.com/mupq/pqm4.

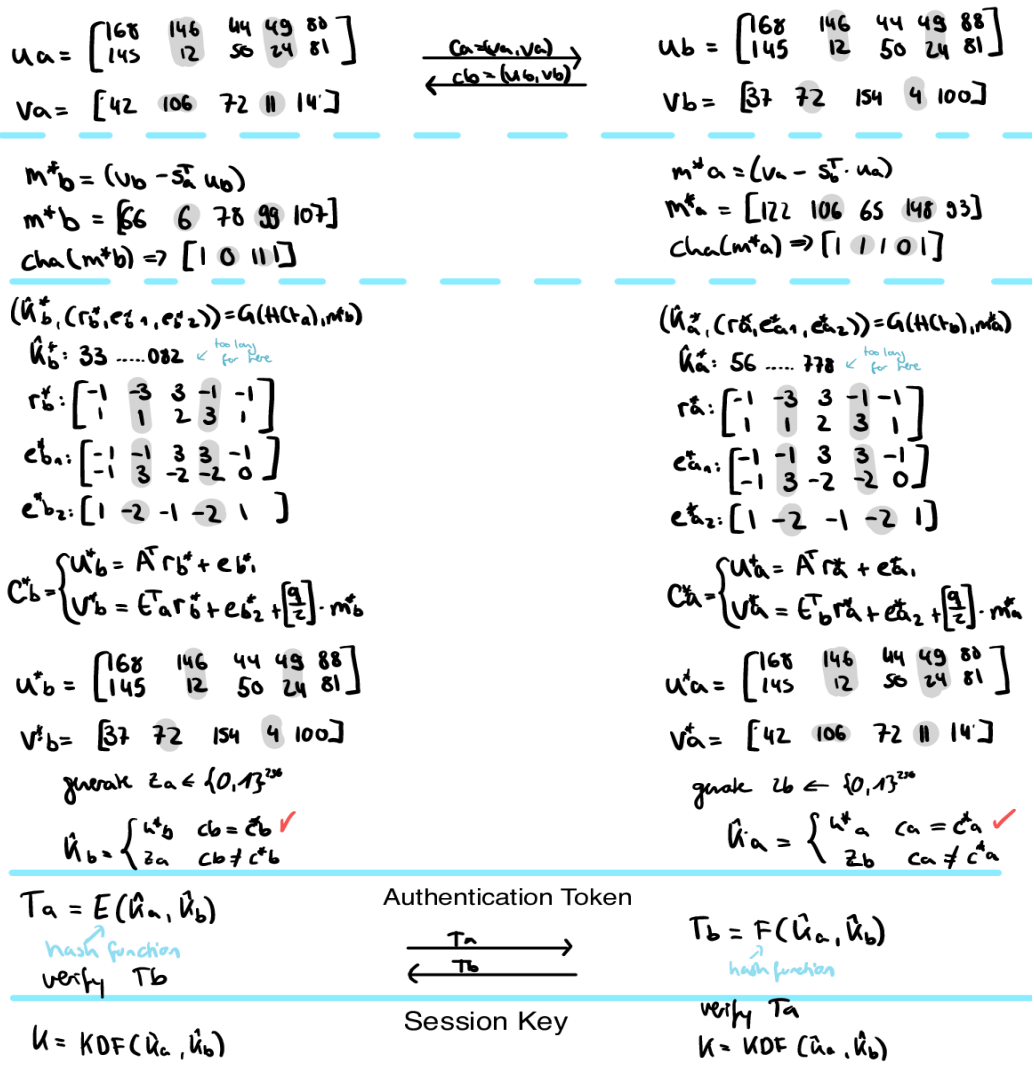
[7] NIST. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. Tech. rep. [Online; accessed 18. Jul. 2022]. July 2022. URL: https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf.

[8] *SystemGroups - Debian Wiki*. [Online; accessed 17. Jul. 2022]. July 2022. URL: https://wiki.debian.org/SystemGroups.

[9] *Users and groups - ArchWiki*. [Online; accessed 17. Jul. 2022]. July 2022. URL: https://wiki.archlinux.org/title/Users_and_groups.

A Kyber-Ding-PACE on Paper




$u_a = \begin{bmatrix} 168 & 146 & 44 & 49 & 88 \\ 145 & 12 & 50 & 24 & 81 \end{bmatrix}$
$v_a = [42\ 106\ 72\ 11\ 14]$
$c_a = (u_a, v_a)$
$c_b = (u_b, v_b)$
$u_b = \begin{bmatrix} 168 & 146 & 44 & 49 & 88 \\ 145 & 12 & 50 & 24 & 81 \end{bmatrix}$
$v_b = [37\ 72\ 154\ 4\ 100]$
$m^*_b = (v_b - s_a^T \cdot u_b)$
$m^*_b = [56\ 6\ 78\ 98\ 107]$
$cha(m^*_b) \Rightarrow [1\ 0\ 1\ 1\ 1]$
$m^*_a = (v_a - s_b^T \cdot u_a)$
$m^*_a = [122\ 106\ 65\ 148\ 93]$
$cha(m^*_a) \Rightarrow [1\ 1\ 1\ 0\ 1]$
$(\hat{k}^*_b, (r^*_b, e^*_{b1}, e^*_{b2})) = G(H(r_a), m^*_b)$
$\hat{k}^*_b$: 33 ..... 082 (too long for here)
r^*_b: $\begin{bmatrix} -1 & -3 & 3 & -1 & -1 \\ 1 & 1 & 2 & 3 & 1 \end{bmatrix}$
e^*_{b1}: $\begin{bmatrix} -1 & -1 & 3 & 3 & -1 \\ -1 & 3 & -2 & -2 & 0 \end{bmatrix}$
e^*_{b2}: $[1\ -2\ -1\ -2\ 1]$
$C^*_b = \begin{cases} u^*_b = A^T r^*_b + e^*_{b1} \\ v^*_b = E_a^T r^*_b + e^*_{b2} + \lfloor \frac{q}{2} \rfloor \cdot m^*_b \end{cases}$
$u^*_b = \begin{bmatrix} 168 & 146 & 44 & 49 & 88 \\ 145 & 12 & 50 & 24 & 81 \end{bmatrix}$
$v^*_b = [37\ 72\ 154\ 4\ 100]$
generate $z_a \in \{0,1\}^{256}$
$\hat{k}_b = \begin{cases} k^*_b & c_b = c^*_b \\ z_a & c_b \neq c^*_b \end{cases}$
$(\hat{k}^*_a, (r^*_a, e^*_{a1}, e^*_{a2})) = G(H(r_b), m^*_a)$
$\hat{k}^*_a$: 56 ..... 778 (too long for here)
r^*_a: $\begin{bmatrix} -1 & -3 & 3 & -1 & -1 \\ 1 & 1 & 2 & 3 & 1 \end{bmatrix}$
e^*_{a1}: $\begin{bmatrix} -1 & -1 & 3 & 3 & -1 \\ -1 & 3 & -2 & -2 & 0 \end{bmatrix}$
e^*_{a2}: $[1\ -2\ -1\ -2\ 1]$
$C^*_a = \begin{cases} u^*_a = A^T r^*_a + e^*_{a1} \\ v^*_a = E_b^T r^*_a + e^*_{a2} + \lfloor \frac{q}{2} \rfloor \cdot m^*_a \end{cases}$
$u^*_a = \begin{bmatrix} 168 & 146 & 44 & 49 & 88 \\ 145 & 12 & 50 & 24 & 81 \end{bmatrix}$
$v^*_a = [42\ 106\ 72\ 11\ 14]$
generate $z_b \leftarrow \{0,1\}^{256}$
$\hat{k}_a = \begin{cases} k^*_a & c_a = c^*_a \\ z_b & c_a \neq c^*_a \end{cases}$
$T_a = E(\hat{k}_a, \hat{k}_b)$
hash function
verify T_b
Authentication Token
T_a
T_b
$T_b = F(\hat{k}_a, \hat{k}_b)$
hash function
verify T_a
Session Key
$k = KDF(\hat{k}_a, \hat{k}_b)$
$k = KDF(\hat{k}_a, \hat{k}_b)$

Acronyms

CTS clear to send. 22

GND ground. 22

RTS request to send. 22

RXD recieve data. 22

TXD transmit data. 22

USART Universal synchronous/asynchronous receiver/transmitter. 22

USB universal serial bus. 22