B-PSE Project Report: Implementation of eID Protocols - SS23

Tobias Depuydt-Wiedemann[1], Yahya El Hadj Ahmed[1], Patrick Fender[1], Gisane Gasparyan-Jung[1], David Haas[1], Yannick Lechler[1], Felix Maximilien Ehondje Ndoumbe[1], and Rafael Cabral Vogt[1]

Darmstadt University of Applied Sciences
{david.c.haas, rafael.vogt, patrick.fender, gisane.gasparyan, felix.m.ndoumbe, yahya.ahmed, tobias.wiedemann, yannick.lechler}@stud.h-da.de

1 Introduction

1.1 Project Description

This project aims to implement and evaluate a quantum-resistant version of the eID (electronic identification) and eMRTD (electronic Machine Readable Travel Document) security protocols known as PACE (Password Authenticated Connection Establishment) and EAC (Extended Access Control). The purpose is to enhance the security of these protocols to withstand potential attacks from future quantum computers.

Quantum-resistant security protocols primarily rely on post-quantum cryptography (PQC) schemes. These schemes can be integrated into security protocols as replacements for classical schemes, ensuring resistance against attacks by quantum computers.

The PACE protocol, which is the main focus of this project, currently employs the Diffie-Hellman (DH) key agreement scheme, based on the hardness of the discrete logarithm problem. The objective is to incorporate a new type of scheme called PQC KEM (Key Encapsulation Mechanism) to replace the DH key agreement. However, this requires modifying the protocol's design and conducting performance and security testing on the new version.

1.2 Goals

The main goals for this semester's project were:

- Implementing the PAKEM (Password Authenticated Key Encapsulation Mechanism) based on an existing implementation from previous student projects in this repository.
- Integration of the USART communication library libOpenCM3 for STM32 boards.
- Integration of an optimized implementation of CRYSTALS-Kyber-KEM from the pqm4 library.

- Implementation of performance benchmarks to analyze time and memory usage.
- Optional: Integration of an existing PQC EAC implementation from Margraf, Morgner, Fischlin, or other sources.
- Optional: Integration of SABER-KEM and FrodoKEM as alternative options to Kyber-KEM.

These additional goals provided opportunities to enhance the functionality and security of the project. However, the primary focus for this semester was the implementation of PAKEM as the central encryption mechanism.

2 Preliminaries

At the beginning of the project, we dedicated several weeks to delving into the theoretical foundations. Our main focus was to familiarize ourselves with various concepts and terminologies that were relevant to our project. This theoretical phase was crucial as it provided us with a strong understanding and a solid foundation for the practical implementation of our project.

2.1 Basic Cryptography

As a starting point, we began by familiarizing ourselves with basic cryptography concepts. We delved into the fundamentals of symmetric and asymmetric cryptography. Particularly, we emphasized asymmetric cryptography as we intended to utilize the Key Encapsulation Mechanism (KEM) in our project, which is also employed in PAKEM.

Additionally, we familiarized ourselves with cryptographic hash functions. We learned about password-authenticated public-key encryption (PAPKE) and gained understanding of Ding-PAKE. These concepts were crucial as our objective was to incorporate them into the implementation of PAKEM. By leveraging these concepts, we aimed to replace the existing mechanism through the implementation of PAKEM.

Furthermore, we explored the concepts of chosen-ciphertext attack (CCA) and chosen-plaintext attack (CPA) security. We also delved into encrypted key exchange (EKE). These topics deepened our comprehension of secure encryption methods and provided valuable insights for utilizing and applying PAKEM.

Key Encalpsulation Mechnanism (KEM) A Key Encapsulation Mechanism (KEM) is a cryptographic method used for securely transmitting a session key between two parties using public key encryption. In traditional public key encryption (PKE), encrypting longer messages can be tedious, so a KEM is employed to encrypt the session key with the recipient's public key. Similar to PKE, the encrypted session key can be decrypted using the corresponding private key. The transmitted session key is then used for symmetric encryption and decryption of messages exchanged between the parties.

A KEM involves three fundamental algorithms:

- **Key generation algorithm:** This algorithm generates a pair of public and private keys.
- **Encryption algorithm:** Given a plaintext message and the recipient's public key, this algorithm produces a ciphertext along with the session key.
- **Decryption algorithm:** This algorithm takes the ciphertext and the recipient's private key to compute the session key.

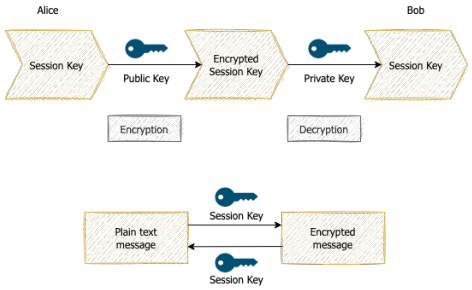

Fig. 1. KEM
[cod22]

Due to the limited length of asymmetric keys, a KEM employs an algorithm to generate a random element in the underlying finite group of the public key system. The symmetric key is derived by hashing this element, eliminating the need for padding.

By utilizing a KEM, Alice and Bob can establish a shared session key, which enables them to securely encrypt outgoing messages and decrypt incoming messages from each other. This approach combines the advantages of asymmetric cryptography (such as key distribution) with the efficiency of symmetric encryption for message transmission.[cod22]

2.2 Post Quantum Cryptography

Post-Quantum Cryptography (PQC) is a branch of cryptography that focuses on developing cryptographic algorithms designed to resist attacks by quantum computers. Traditional cryptographic algorithms such as RSA and ECC rely on mathematical problems like factorization, discrete logarithms, and elliptic curve discrete logarithms. However, these problems can be efficiently solved by powerful quantum computers, rendering these algorithms insecure.

To address this challenge, post-quantum algorithms have been proposed as potential solutions. During our studies, we explored different types of post-quantum cryptography algorithms, including:

Lattice-based cryptography Lattices have become a crucial component in modern cryptography due to their resistance against sub-exponential and quantum attacks. These mathematical structures, first studied by mathematicians

like Lagrange and Gauss, provide a strong periodicity property. In cryptography, a lattice is a set of points in n-dimensional Euclidean space, and any element can be uniquely represented as a linear combination of a basis set of vectors with integer coefficients. However, in practical cryptographic systems, which require finite representations such as bit strings, it is necessary to select finite elements from the infinite lattice structures. This ensures compatibility with cryptographic operations while leveraging the security properties offered by lattices.[RK19]

- *Kyber-KEM:* Kyber has been selected by NIST as a post-quantum secure public key encryption (PKE) and key exchange mechanism (KEM) under NIST's post-quantum cryptography (PQC) standardization initiative.
  Kyber offers both IND-CPA-secure public key encryption [Kyber CPAPKE] and IND-CCA2-secure key encapsulation mechanism [Kyber CCAKEM]. Its security is based on the hardness of solving the learning-with-errors (LWE) problem in structured lattices.
  In the case of IND-CPA-secure Kyber PKE, two communicating parties generate their own key pairs and exchange their public keys. Using the recipient's public key, a fixed-length message (32 bytes) can be encrypted. The ciphertext can be decrypted using the corresponding secret key (owned privately by the key owner), allowing the recovery of the 32-byte message.
  On the other hand, in the case of IND-CCA2-secure Kyber KEM, two parties interested in secure communication over a public and insecure channel can generate a shared secret key of arbitrary byte length. This shared secret key is derived from a key derivation function (KDF), specifically SHAKE256 XOF in this context. Both parties obtain the same shared secret by seeding SHAKE256 XOF with a common secret. This secret is 32 bytes long and is communicated from the sender to the receiver using the underlying Kyber PKE mechanism[BDK$^+$18b]
  However, one disadvantage of Kyber is that its public module 'a' consists of k^2 polynomials, which is k times larger compared to Ring Learning with Errors (RLWE) with similar security. Despite this drawback, Kyber remains a promising choice for post-quantum cryptography due to its robustness against attacks from powerful quantum computers.
- *Saber-KEM:*Saber has been selected as a finalist in the NIST PQ standardization project. Its security relies on the difficulty of the Module Learning with Rounding problem (MLWR). Saber begins with an IND-CPA secure encryption scheme called Saber.PKE, and further presents an IND-CCA secure key encapsulation mechanism (KEM) known as Saber.KEM. The Saber.KEM is obtained from Saber.PKE through a version of the FO transform.[NDGJ21]
- *Frodo-KEM:* Frodo is a post-quantum cryptography scheme based on the Learning with Errors (LWE) problem. The elements of martix are sampled from a discrete Gaussian distribution. Frodo-KEM utilizes different parameter sets for different security levels: (640, 215, 12, 2.8), (976, 216, 10, 2.3), and (1344, 216, 6, 1.4) for Frodo-640, Frodo-976, and Frodo-1344 respectively. The three main algorithms that constitute Frodo-KEM are key generation (KeyGen), encapsulation (Encaps), and decapsulation (Decaps).[LBB22]

Multivariate cryptography: This type of cryptography involves using multivariate polynomial equations to provide security against quantum attacks. Multivariate cryptographic algorithms offer resistance to quantum attacks due to the computational hardness of solving systems of multivariate equations.

Hash-based cryptography: In 1979, Ralph Merkle introduced the first hash-based digital signature scheme, which relies on the security of one-way hash functions. Despite the limitation of producing a fixed number of signatures at once, these schemes offer long-term security against known quantum computer algorithms. In 1990, Merkle developed the Merkle signature scheme, which enables the conversion of a one-time signature into a multi-time signature by utilizing the Lamport-Diffie one-time signature as a foundational component.[RK19]

Supersingular elliptic curve isogeny cryptography: This cryptographic approach is based on isogenies, which are mappings between elliptic curves. Supersingular elliptic curve isogeny cryptography offers a different mathematical foundation for providing post-quantum security.

Code-based cryptography: This type of cryptography relies on error-correcting codes to provide security against quantum attacks. Code-based cryptographic algorithms are based on the hardness of decoding certain linear codes.

For our project, it was particularly important to understand lattice-based cryptography, as we integrated the optimized CRYSTALS-KYBER algorithm. Kyber is a post-quantum Key Encapsulation Mechanism (KEM) that builds upon the Learning with Errors (LWE) problem. It utilizes a ring structure that is similar to the one used in the NewHope algorithm.

2.3 PAKEM(Password Authenticated Key Encapsulation Mechanism)

Pakem is a cryptographic mechanism that enables secure data exchange between two instances, namely the client and the server. Both the client and the server apply the selected key derivation function to the password, with the client also generating a key pair using the specified KEM. The client then constructs its authenticated public key by encrypting the public key with the shared derived encryption key and sends it to the server, where it is decrypted using the server-side shared derived encryption key.

Figure 2 shows the protocol diagram for PAKEM.

Alice		**Bob**
$Password\ \pi$		$Password\ \pi'$
$K_\pi = \mathcal{KDF}(\pi)$		$K_{\pi'} = \mathcal{KDF}(\pi')$
$sk_a, pk_a \xleftarrow{\$} KeyGen$		
$apk_a \xleftarrow{\$} C_{K_\pi}(pk)$		
	$\xrightarrow{apk_a}$	
		$pk'_a = \mathcal{C}^{-1}_{K_{\pi'}}(apk_a)$
		$(c_b, \overline{K}) = Encap(pk'_a)$
	$\xleftarrow{c_b}$	
$\overline{K}^* = Decap(sk_a, c_b)$		
$K = \mathcal{KDF}(\overline{K}^*)$		$K = \mathcal{KDF}(\overline{K})$

Fig. 2. PAKEM

3 Implementation

3.1 Implementation of PAKEM

New Code structure Starting out the project, it soon became clear that we would have to come up with a new program infrastructure. The old implementation was entirely based on Kyber [BDK+18a] and was therefore not designed to be modular or flexible in any way. That is why a lot of time went into finding a new structure that would achieve these goals, with the bonus tasks of improving memory efficiency and usability in general in the back of our heads.

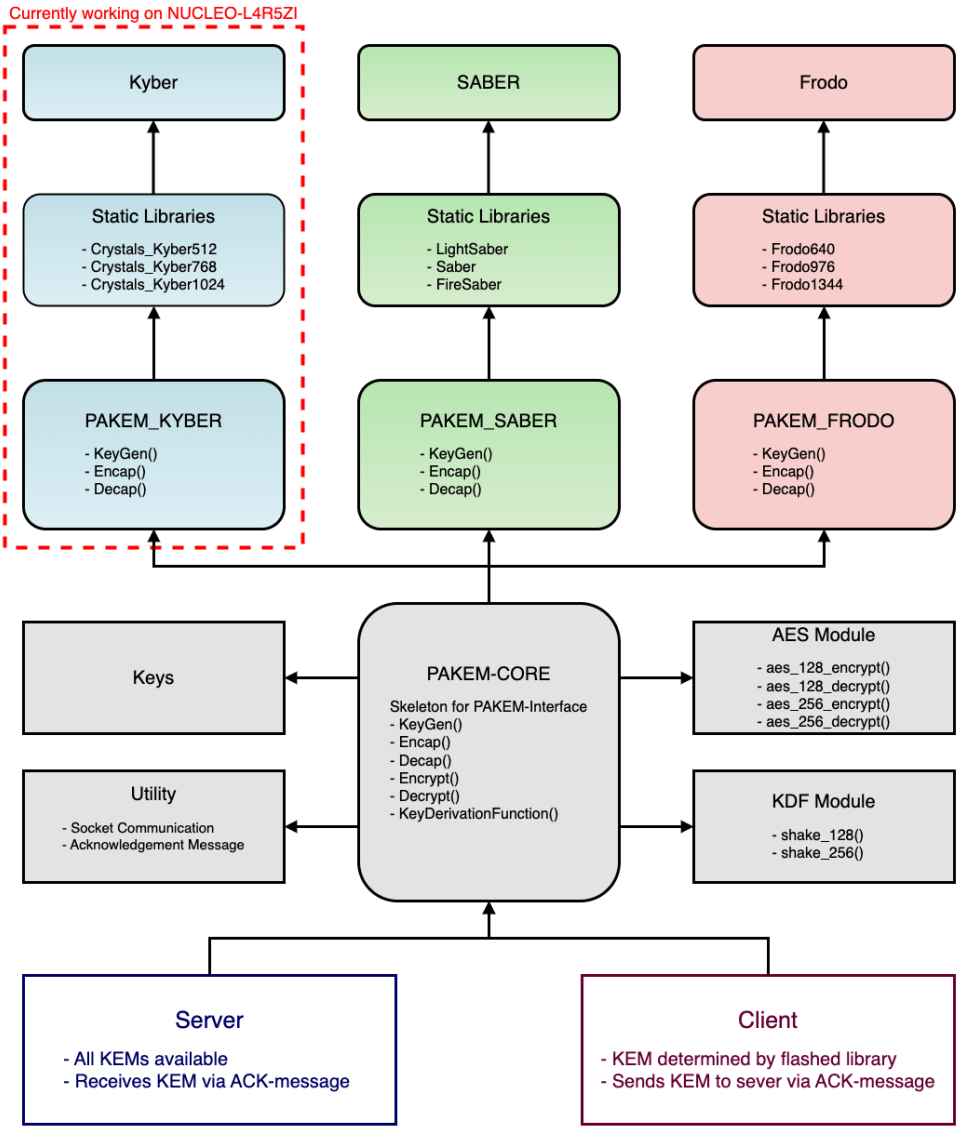

Fig. 3. Program Architecture

At the very heart of our code sits the PAKEM-Core, an abstraction layer that provides an interface, that can be assigned concrete functions dynamically. While we were initially only planning to assign the key encapsulation mechanism dynamically, the security levels of the AES encryption and hash-based key derivation can now also be asserted. Since Kyber, SABER [DKRV18] and Frodo [BCD+16] all come with 3 different security levels themselves, our program allows to specify which level is to be used.

In order to make the PAKEM implementation as user friendly as possible, we decided early on to work with preprocessor variables as much as possible. In the current state of the implementation, a user does not have to interact with the code itself in order to run the PAKEM protocol. Additionally we have adapted the message, that initiates the protocol. Instead of a random payload, the client sends the server an unsigned integer. This integer contains the used KEM as well as the security level, as well as the security level of the AES-encryption and the hashing. We achieve this by using a simple binary encoding, the server decodes the message upon receiving and includes the necessary static libraries to match the client.

Another important point we tackled was the memory-efficiency of the project. The old implementation contained a large amount of redundant memory structures and memory leaks. Therefore, we have reworked the way that keys are stored over the course of the protocol and implemented a singular structure, that contains all the keys, as well as methods for appropriate memory management. To test our work, we used Valgrind as a tool to benchmark memory efficiency and to iron out any remaining memory leaks.

Integrating the different KEMs Each KEM requires specific compiler flags for optimal execution, and they often operate at different security levels. By encapsulating these complexities in static libraries, multiple benefits are achieved:

- Flag Management: KEMs requires special compiler flags to ensure the optimal execution of the core cryptographic operations. Using static libraries we eliminated the need to manage the complex flags in the main codebase, resulting in cleaner and more comprehensible compilation process.
- Modularity and Maintainability: static libraries encapsulates the algorithm-specific optimizations in isolated units. This modularity simplifies maintenance, future enhancements to the project and promotes code stability.
- Code Duplication: As different KEMs may demand separate source code to meet their unique security requirements, code duplication wouldn't be avoided. By encapsulating the algorithm-specific code optimizations and security parameters within static libraries, we eliminate the need to duplicate source code for each security level. This reduction in code duplication simplifies the project structure, leading to a more manageable and maintainable codebase.

The static libraries were created within separate Docker containers and subsequently integrated into the application within the virtual environment. However,

for the hardware environment, we currently use the source code of the algorithms to guarantee a more effective debugging.

The integration of SABER was omitted due to its dependency on OpenSSL. The complexity of compiling OpenSSL for ARM and the probably not successful integration given the limited hardware resources deemed the approach to be impractical. This led to the decision of not including SABER in the implementation to ensure project deadlines are met.

Similarly, the integration of FRODO posed a challenge as it requires an operating system environment (Unix, Windows), which is absent on the hardware. In the last week we found an ARM-compiled version of FRODO. However, we made the decision to not integrate it due to time constraints.

Initialising the communication As mentioned above, everything starts with the customer. To initiate communication, the client selects its operating mode using the Acknowledgement Message, in which it encodes the security level of the selected kem, AES (Advanced Encryption Standard) and shake in an unsigned integer with binary correspondence. The server will then receive this information via the coded message and will check whether it has the necessary libraries before being able to place its PAKEM-core function pointers accordingly.

```
1 void sendClientPakemSetup(struct CLIENT_ACK_MESSAGE
    clientAckMessage) {
2     struct ByteArray ackMessageEncoded = encode(
    clientAckMessage);
3     sendData(clientSocket, ackMessageEncoded);
4     struct ByteArray ackMessage = receiveData(clientSocket);
5     printf("%s\n", ackMessage.data);
6     free((void *) ackMessage.data);
7 }
```

Listing 1.1. initialising communication from the client-side

Initialising the communication As mentioned above, everything starts with the customer. To initiate communication, the client selects its operating mode using the Acknowledgement Message, in which it encodes the security level of the selected KEM, AES (Advanced Encryption Standard) and SHAKE in an unsigned integer with binary correspondence. The server will then receive this information via the coded message and will check whether it has the necessary libraries before being able to place its PAKEM-core function pointers accordingly.

```
1 void sendClientPakemSetup(struct CLIENT_ACK_MESSAGE
    clientAckMessage) {
2     struct ByteArray ackMessageEncoded = encode(
    clientAckMessage);
3     sendData(clientSocket, ackMessageEncoded);
4     struct ByteArray ackMessage = receiveData(clientSocket);
```

```
    printf("%s\n", ackMessage.data);
    free((void *) ackMessage.data);
}
```

Listing 1.2. initialising communication from the client-side

```
struct PAKEM_UNIT * checkClientPakemSupported(struct
    CLIENT_ACK_MESSAGE clientAckMessage, struct PAKEM_UNIT **
    pakemsArr) {
    struct PAKEM_UNIT *pakemUnit = pakemsArr[
    getPakemSecLevelIndex(clientAckMessage.kem,
    clientAckMessage.level)];
    bool notSupported = (pakemUnit == NULL);
    if (notSupported) {
        fprintf(stderr, "Unsupported Pakem KEM:%d, SEC_LEVEL
    :%d. exit", clientAckMessage.kem, clientAckMessage.level)
    ;
    }
    char *unsupportedPakem = (notSupported) ? "UNSUPPORTED
    PAKEM\0" : "OK\0";
    struct ByteArray ackMessageServer = {(uchar *)
    unsupportedPakem, strlen(unsupportedPakem)};
    sendData(serverSocket, ackMessageServer);
    if (notSupported) exit(-1);
    return pakemUnit;
}
```

Listing 1.3. setting PAKEM_UNIT pointer according to PAKEM supported by the client

Key exchange The following step then consists in both client and server deriving a shared password from an identical pin they both know initially - if they are not corrupted. For this end, both client and server make use of the SHAKE KDF (key derivation function) and store the resulting password using a `struct Key` pointer.
The client continues by generating a key pair (one private and one public key). To perform this, it uses a key generation function that is derived from the information on the supported PAKEM. The public key is then encrypted using AES in CBC mode with the shared derived password. The AES security level (128 or 256 bit) is chosen according to the supported PAKEM.

```
void generateKeyPair(KeyGenFunc keyGenFunction, struct
    ByteArray *publicKey, struct ByteArray *privateKey) {
    allocate2Data(privateKey, publicKey);
    keyGenFunction(publicKey->data, privateKey->data);
}
```

Listing 1.4. client generating a key pair using a configurable key generation function

Authentication and encapsulation Once the server receives the client's encrypted public key, it decrypts it using a decryption function that is chosen according to the information on the supported PAKEM received in the Acknowledgement Message. The shared password serves as key for the AES decryption. Decryption success is thus dependent on both the client's and the server's knowledge of the correct initial pin and can thus be viewed as a means of authentication.
The server then proceeds to encapsulate the key, using the encapsulation function that corresponds to the chosen PAKEM and the previously decrypted public key received from the client. As results, the encapsulation yields both a ciphertext and a shared secret. Both are stored using the `struct Key` pointer. The server then sends the ciphertext to the client.

```
1  void sendEncapsulateSecret(struct ByteArray *cipherServer,
       struct ByteArray *sharedSecret,
2                         struct ByteArray *publicKeyClient,
       EncapFunc encapFunction) {
3      allocate2Data(cipherServer, sharedSecret);
4      encapFunction(cipherServer->data, sharedSecret->data,
       publicKeyClient->data);
5      sendData(serverSocket, *cipherServer);
6      printf("Cipher:\n");
7      printHex(cipherServer->data, cipherServer->length);
8      printf("\n");
9      deallocate2Data(cipherServer, publicKeyClient);
10 }
```

Listing 1.5. server encapsulating the client's public key and sending the resulting ciphertext to the client

Decapsulation and derivation of a symmetric key When the client receives the ciphertext the server just sent to it, it goes on by decapsulating the shared secret using the client's private key as well as a decapsulation function that is set according to the supported PAKEM. The shared secret is stored using the struct Key pointer.
Now that both server and client have the shared secret, they both derive a session key stored using the same key pointer and the SHAKE KDF. This session key serves as a symmetric key which allows client and server to entertain an AES encrypted communication.

```
1  void receiveDecapsulateSecret(struct ByteArray *cipherServer,
2                            struct ByteArray *sharedSecret,
3                            struct ByteArray *privateKeyClient,
4                            DecapFunc decapFunction) {
5      allocate2Data(sharedSecret, cipherServer);
6      struct ByteArray received_cipher = receiveData(
       clientSocket);
7      memcpy(cipherServer->data, received_cipher.data,
       received_cipher.length);
```

```
    deallocate1Data(&received_cipher);
    decapFunction(sharedSecret->data, cipherServer->data,
    privateKeyClient->data);
    deallocate2Data(cipherServer, privateKeyClient);
}
```

Listing 1.6. client decapsulating the ciperhtext sent to it by the server

Demonstration Our code includes a demonstration of the PAKEM implementation in a virtual environment which serves to give an immediate feedback on whether the code works as it should. Both client and server call a function `demonstrateItWorked` which emulates a complete communication between each other where the outcome of every step is printed in the terminal so that programmers can check the code's functionality.

```
void demonstrateItWorked(EncryptFunction encryptFunction,
    DecryptFunction decryptFunction,
                        struct ByteArray *sessionKey) {
    char *message = "Hello Bob, my name is Alice and I study
    Computer Science at the Hochschule Darmstadt\0";
    struct ByteArray message_byte = {.data = (uchar *)
    message, .length = strlen(message)};
    const struct ByteArray *encrypted_msg_data =
    encryptFunction(sessionKey, &message_byte);
    sendData(clientSocket, *encrypted_msg_data);
    freePaddingResult(encrypted_msg_data);
    struct ByteArray encMessage = receiveData(clientSocket);
    struct ByteArray *receivedMessage = decryptFunction(
    sessionKey, &encMessage);
    printf("\n");
    printf("%s", receivedMessage->data);
    printf("\n");
    freeUnPaddingResult(receivedMessage);
    free((void *) encMessage.data);
}
```

Listing 1.7. demonstration of the code's functionality from the cleint side

3.2 Integration of the USART communication library libOpenCM3 for STM32 boards

Objective: Our goal for this project was to integrate the communication with libOpenCM3 [lib] while preserving the STM32 HAL (Hardware Abstraction Layer) for other setups, including GPIO (General-Purpose Input/Output) and Clocks.

Progress and Challenges: We successfully managed to link and compile with libOpenCM3, demonstrating the interoperability of these two frameworks at a

basic level. In this stage we were able to send messages to the server application. Within this process we faced problems with the communication protocol of the server, the integrated development environment as also with initializing the hardware. To look at the those problems separately we decided to start developing basic programs from scratch and as the progress continues to integrate the new functionalities step by step. With time as our most shrinking resource we were unable to establish a fully functional USART communication.
First approach: Our first approach was to identify every aspect of the HAL code and to tried to free the existing code base of it. Therefore we documented the functionality and appearance of the existing HAL code. We came to the conclusion that this approach would be too time consuming and the existing interaction with the HAL library was integrated to deep into the existing program.
Second approach: To get a better understanding of how the libOpenCM3 works we started developing a hello world program where we tried so send a 'Hello world' string to the server. To analyze the communication traffic we used a program called hterm [hte] which made it possible to see what the server obtains. Here we encountered the first bigger problem. Instead of the server receiving the right message it interpreted the message as random bits and not the 'Hello world' string we tried to send. We suspected that the initial setup of the clock, USART and GPIO were the root cause of the problem.
Third approach: Due to our missing understanding of the possible interference of the setup with the communication we chose to orient on an existing approach from a former developer. We modified the code and made it work for our purposes. As this was done we went back to our former approach and used the new code base to implement the functionalities we need step by step. We were now still not able to send a string to the server and get it interpreted right. As we continued to work on the receiving function we also failed on this task. The main problem in this case was that debugging the interrupt service routine led to break the debugging process. With this issue and our restricted knowledge we were not able to further investigate on the problems. So we decided to document our steps and ask for support from more experienced developers in this field. The response time exceeded our available time.

Future Steps: A deeper investigation is required to fully understand and resolve the observed issues. These investigations should focus on how the two frameworks, libOpenCM3 and STM32 HAL, interact and potentially conflict, especially with respect to USART communication and interrupt handling. Furthermore, exploring more robust and effective debugging tools might aid in dissecting the complications inherent in our current setup. Also a documentation of the existing communication protocols should be created.

3.3 Evaluation of Implementations

Tested System Configurations The memory and speed benchmarking has been performed on the following system configurations:

(1) **Modified Kyber-Ding-PACE** (branch `feature/pace-benchmarks-memory`, commit `be8cb857`): This is the project configuration from before we started work in SS23 (Apr 2023)
(2) **PAKEM with pqm4 speed Kyber Level 2, first working version** (branch `feature/pakem-benchmarks-memory`, commit `9ff466f4`): Our implementation of the PAKEM protocol with the pqm4 speed implementation of Kyber Level 2 (kyber512) as well as 128bit-AES and 256bit-shake. In this configuration, the memory allocation was not optimal. Deallocation was missing in many places. For the following configurations we improved the memory usage.
(3) **PAKEM with pqm4 speed Kyber Level 2** (branch `feature/pakem-benchmarks-memory`, commit `9ff466f4`): This configuration implements the PAKEM protocol with the pqm4 speed implementation of Kyber Level 2 (kyber512) as well as 128bit-AES and 256bit-shake.
(4) **PAKEM with pqm4 speed Kyber Level 3** (branch `feature/pakem-benchmarks-memory`, commit `9ff466f4`): This configuration implements the PAKEM protocol with the pqm4 speed implementation of Kyber Level 3 (kyber786) as well as 128bit-AES and 256bit-shake.
(5) **PAKEM with pqm4 speed Kyber Level 4** (branch `feature/pakem-benchmarks-memory`, commit `9ff466f4`): This configuration implements the PAKEM protocol with the pqm4 speed implementation of Kyber Level 4 (kyber1024) as well as 128bit-AES and 256bit-shake.

The following sections focus on the differences between the final PAKEM and Modified Kyber-Ding-PACE implementations. The full results and discussion, including the unoptimized Kyber (configuration 2), can be found in the wiki of the GitLab project.

Memory Benchmarking Implementation Since no tool was found in our research that could provide the benchmarking functionality on the board, we opted to implement our own solution. The memory benchmarking implementation includes several components and functionalities:

(1) **Memory Tracking Functions:** The implementation includes wrapper functions for `malloc`, `free`, and `calloc`, which intercept standard memory allocation and deallocation calls to track and measure heap memory usage. These functions update memory usage metrics such as current heap usage, total heap usage, and peak heap usage. The `malloc` and `calloc` wrappers reserve more memory than initially requested to store the size information as well, but only track the initially requested amount of memory as to not alter the benchmark. The reason for this is that the `free` wrapper is provided the information of how much memory was freed and can keep track of this correctly.

(2) **Memory Usage Tracking:** The implementation maintains a `MemoryUsage` struct to track memory usage. This struct holds metrics such as total memory, current memory, global peak memory, and peak memory for specific subroutines.
(3) **Memory Benchmarking Functions:** The implementation provides functions for benchmarking memory usage:
   - `transmitMemoryProfile`: This function transmits memory profile information to the server. It formats memory usage metrics, including total memory, peak memory, and memory deltas, into a message buffer and sends it to the server.
   - `reset_peak_memory_usage`: Used before measuring a subroutine to reset the 'local' peak (the global peak remains unchanged).
   - `get_peak_memory_usage`: Used after measuring a subroutine to get the peak since the last reset.

Memory Metrics The memory benchmarking implementation includes several metrics to capture different aspects of memory usage. These metrics are stored in the `MemoryUsage` struct and provide insights into memory allocation patterns:

`total (bytes)` Total heap memory allocated throughout the program.
`global_peak (bytes)` Highest heap memory used at any point.
`beforeProtocol, afterProtocol (bytes)` Memory usage before and after the protocol execution, respectively.
`keygen_peak, keygen_delta (bytes)` Peak and change in memory during key generation.
`apkEncrypt_peak, apkEncrypt_delta (bytes)` Metrics related to AES encryption.
`decap_peak, decap_delta (bytes)` Peak and change in memory during decapsulation.

Additional Measurements and Analysis In addition to memory benchmarking, the following measurements and analysis have been performed:

- **RAM and FLASH Usage:** The CubeIDE Build Analyzer has been used to obtain information about RAM and FLASH usage. This helps in understanding the program's memory footprint.
  $FLASH$ = code (text) + initialized global variables (copied from FLASH to RAM during startup).
  RAM = initialized + uninitialized global variables.
- **Speed Benchmarking:** Every function has been tested for speed benchmarking. A timer class was inserted in the STM32-Client Project. The code for the speed benchmarking can be found in the branch *feature/pakem-benchmarkings-time/Sources/stm32_client*. The timer commands used in the project are as follows:
  - INIT_TIMER: Initiate the timer protocol

- RESET_TIMER: Reset the timer to 0.
- START_TIMER: Timer start.
- STOP_TIMER: Stop the timer.
- GET_RESULT_MS: Convert the result in Milliseconds
- CONCLUDE_TIMING: Stops the running timer and resets it

Steps for Reproducing Memory Benchmark Results To reproduce the benchmarking results, follow these steps:

1. Clone the repository and checkout the specific commit used as the basis for benchmarking the specific configuration (see configurations above).
2. Since CubeIDE doesn't behave well when importing projects, you need to create a new STM32 project (any name), then replace all the files with the files from the correct folder in the repo, and finally rename the .ioc file to the name you gave your new STM32 project. The correct folders to copy into the CubeIDE project are:
   (a) for Modified Kyber-Ding-PACE: import `Sources/PACE-client-mem-benchmark`
   (b) for PAKEM with pqm4 speed Kyber Level 2, unoptimized first working version: `import Sources/PAKEM-client-unoptimized-lvl2-memory-benchmark`
   (c) for the other PAKEM configurations: import the `Sources/PAKEM-client-lvlX-memory-benchmark` folder with the specific Kyber security level
3. For PAKEM: add preprocessor flags. These should already be imported correctly. If not, add `USE_KEM=KYBER`, choose `KYBER_K=?`, `KYBER_?=1` and `SEC_LEVEL=?` according the the Kyber security level, and choose `AES_LEVEL=?` and `SHAKE_LEVEL=?` as desired.
4. Add benchmarking linker flags. These should already be imported correctly. If not, add these to 'Other Linker flags': `-Wl,-wrap,malloc -Wl,-wrap,free -Wl,-wrap,calloc`
5. Connect the board and run the project using CubeIDE.
6. Build and start the server.
   (a) For PAKEM:
      – Make sure to build for the Kyber level the client uses, or all versions by specifying `cmake -DKYBER=234 -DVIRTUAL=0 .`
      – Run `make`
      – Run the server: `sudo ./server /dev/ttyUSB0`
   (b) For PACE:
      – Use CLion to edit the Run Configuration (Run with argument `/dev/ttyUSB0` and root privileges)
7. Press the reset button (black button) on the board, the server should start printing output to the console
8. Find the measured memory usage in the server console output as well as in the file written to `cmake-build-debug/Sources/virtual-environment/server/memory_benchmarks.txt`
9. Analyze the generated .elf file size (in the Debug folder of the CubeIDE project), RAM and FLASH usage, and stack usage using the provided tools (i.e. CubeIDE Build Analyzer and Static Stack Analyzer).

Results Tables 1 and 2 show the results measured in our benchmarks for memory and speed, respectively.

Discussion of Results for Memory Measurement

1. **Static Memory:** All tested PAKEM configurations use about 0.6% of the 640KB RAM and 4.5-4.6% of the 2MB FLASH on startup, which is less RAM usage but more FLASH usage compared to the original "modified Kyber-Ding-PACE". This suggests that we are using a larger codebase for the PAKEM implementation, but less global variables.
2. **Peak Memory Usage:** The PAKEM implementations demonstrate significant memory usage reduction compared to the previous "modified Kyber-Ding-PACE." At the lowest security level (PAKEM with Kyber Level 2), the peak memory usage (global_peak) is approximately 11% of the previous, while at the highest security level (PAKEM with Kyber Level 4), it is about 28% of the previous. This indicates a substantial reduction in memory consumption across the PAKEM implementations compared to the previous work.
3. **Differences Between Kyber Versions:** The memory usage trend observed across the PAKEM implementations is consistent with the inherent trade-offs in cryptography. As one opts for stronger security levels with larger key sizes or cryptographic parameters, one should expect increased computational and memory requirements. The labels (Kyber512, Kyber768, and Kyber1024) refer to different parameter sets of the Kyber post-quantum key encapsulation mechanism. With larger keys or parameters, cryptographic operations such as key generation, encryption, and decryption can involve more complex mathematical operations or iterations, thus consuming more memory.
4. **Memory Deallocation After Protocol:** The PAKEM implementations exhibit efficient memory deallocation after establishing the secure connection. The allocated memory after connection establishment represents only a fraction of the peak memory usage, amounting to 52%, 58%, and 63% for the three security levels, respectively (given by: afterProtocol / global_peak). Compared to the original "modified Kyber-Ding-PACE," which did not free any memory after connection establishment, the memory optimization in PAKEM implementations is evident.
5. **Memory Usage in Subroutines:** The memory requirements of the measured subroutines vary across the different implementations and security levels. Regarding the differences between delta and peak values, in our PAKEM implementations the delta of each subroutine is always slightly below the corresponding peak, indicating small memory deallocations during each subroutine. The "decap delta" for PAKEM is negative in all three configurations, suggesting a release or reduction of memory during decapsulation - more than what was allocated for decapsulation itself. The decap routine in PACE shows no memory usage at all, potentially because the subroutiens itself does not allocate or free any memory, and so does the function call is makes to

		Previous Work	Our Work		
		Kyber-Ding-PACE	PAKEM Kyber512	PAKEM Kyber768	PAKEM Kyber1024
RAM and FLASH	RAM used (kB)	4.2	3.6	3.6	3.6
	FLASH used (kB)	61.3	91.6	91.9	93.5
Heap	total usage (kB)	48.4	8.8	13.8	19.9
	global peak (kB)	47.3	5.4	8.9	13.3
	before protocol (kB)	0.1	0.2	0.2	0.2
	after protocol (kB)	47.3	2.8	5.2	8.4
	keygen peak (kB)	5.9	3.4	5.6	8.1
	keygen delta (kB)	5.9	3.2	5.4	7.9
	apk encrypt peak (kB)	2.3	1.8	2.5	3.4
	apk encypt delta (kB)	2.3	1.0	1.3	1.8
	decap peak (kB)	0	1.0	2.0	3.4
	decap delta (kB)	0	-1.6	-1.7	-1.5

Table 1. Memory Benchmarks

		Previous Work	Our Work		
		Kyber-Ding-PACE	PAKEM Kyber512	PAKEM Kyber768	PAKEM Kyber1024
Function Calls	sendClientPakemSetup (ms)	504	502	502	502
	generateKeyPair (ms)	67	32	52	82
	sendClientPublicKey (ms)	501	518	525	533
	receiveDecapsulatedSecret (ms)	86	34	56	337
	deriveSessionKey (ms)	45	1	1	1
	total time (ms)	3669	1095	1146	1469

Table 2. Speed Benchmarks

the KEM. The structs for storing the ciphertext, keys and decrypted data have already had memory allocated before this subroutine. This shows a potential flaw in the way memory is measured in our benchmarks: all of the memory a particular subroutine needs is not necessarily allocated inside that subroutine. This can lead to inaccuracies when analysing the memory needs of different subroutines.

Discussion of Results for Speed Measurement

1. **Comparision with the Kyber-DING-PACE Implementation:** The PAKEM protocol stands out from the KYBER-DING protocol for several reasons that contribute to a generally faster implementation. One significant reason for this is the efficiency of the hashing process. In the KYBER-DING protocol, hashing requires more time compared to PAKEM. This is because KYBER-DING employs a more complex hashing procedure that demands more computational power, potentially slowing down the entire encryption process. Another crucial factor is Memory Allocation, the management of memory during implementation. PAKEM utilizes optimized memory allocation, allowing for more efficient memory deallocation. In contrast, the memory allocation in the KYBER-DING protocol might be less optimized, leading to fragmentation and inefficient memory usage. The efficient use of memory by the PAKEM Protocol contributes to speeding up the overall process, as less time is spent managing memory resources.
   Another reason for the shorter time in the PAKEM protocol is that some methods are used multiple times in the Kyber-Ding PACE protocol. Therefore, for example, the `generateKeyPair` and the decap method is almost exactly half the time compared to the Kyber-DING-PACE protocol. The same thig goes to the `deriveSessionKey` method.
2. **Usage of AES and SHAKE:** In the implementation of PAKEM, a notable feature is the ability to apply distinct security levels to various functions within the protocol. SHAKE has been employed for the decapsulation process, while AES serves for key generation. Within SHAKE and AES, one has the flexibility to individually select the desired version, whether it's AES-128 or AES-256, and similarly, SHAKE offers the choice of SHAKE-128 or SHAKE-256. Opting for the 256-bit versions results in longer encryption times compared to the 128-bit variants, which can be seen in the Speed Benchmark tests above. In the benchmark tests conducted, which are illustrated in a graph, AES-128 and SHAKE-128 were utilized. However, the selection of these specific versions is entirely customizable based on requirements and preferences.
3. **Kyber512, Kyber768 und Kyber1024** The implementation provides the flexibility to choose from various security levels within the Kyber protocol as well. These security levels include Kyber512, Kyber768, and Kyber1024. Depending on the specific security level chosen, the corresponding encryption process will require varying amounts of time. It's important to note that the selection of a higher security level, such as Kyber1024, would naturally

demand more computational resources and consequently result in longer encryption times. This adaptive approach allows users to tailor their security requirements to the application's needs while being aware of the trade-offs in terms of processing time and computational overhead.

4. **Decapsulation peak from kyber768 to kyber1024:** A noticable peak can be seen in the `receiveDecapsulatedSecret` method from Kyber768 to Kyber1024 from 56ms to 337ms. This peak could be attributed to several factors: **(1) Resource Consumption**: The use of SHAKE, an extended SHA-3 algorithm, for decapsulation in Kyber1024 can demand higher resource utilization. The increased complexity of this algorithm can contribute to longer processing times. **(2) Memory Requirements**: The larger key size and increased complexity in Kyber1024 can lead to higher memory demands, affecting execution time, especially if memory accesses need optimization. **(3) Implementation Optimization**: It's possible that the decapsulation method for Kyber768 was better optimized than for Kyber1024. A less optimized implementation can result in longer execution times.

4 Conclusion

4.1 Review

The team set out with a challenging yet imperative goal in mind - integrating post-quantum cryptography (PQC) into existing eID and eMRTD security protocols to counter potential threats from quantum computers. Through the course of this project:

- The PAKEM (Password Authenticated Key Encapsulation Mechanism) was successfully implemented on the foundation of previous student projects, using a modular framework for more flexibility and expandability. Notably, the introduction of an ACK-message is a pivotal development that communicates both the KEM and the security levels in use and enables a single eID terminal (server) to communicate with different security configurations across different cards (clients).
- While the integration of the USART communication library libOpenCM3 for STM32 boards was aimed at, it faced substantial hurdles. Despite genuine attempts, integration couldn't be achieved. However, the team has documented the entire process to facilitate future endeavors.
- The integration of the optimized CRYSTALS-Kyber-KEM from the pqm4 library was achieved.
- Performance benchmark frameworks have been implemented using our own solution, allowing the measurement of both memory usage and speed of any function call by instrumenting the code. Benchmarks have been made for some system configurations. In summary, our evaluation highlights the reduction in memory usage and execution time of the PAKEM implementations compared to the original "modified Kyber-Ding-PACE."
- Some goals, though ambitious, could not be achieved within the project's timeframe. Specifically, the integration of an existing PQC EAC implementation and the physical board implementation of SABER-KEM and FrodoKEM.

4.2 Current State

At the culmination of this semester's work:

- The PAKEM protocol has been robustly implemented in the virtual environment and on the board. A demo can be found in the `pakem-demo` branch of the GitLab repo.
- The optimized implementation of CRYSTALS-Kyber-KEM from the pqm4 library has been integrated and each security level runs on the board.
- Although PAKEM with Kyber has been implemented on the board, a persistent issue has prevented its virtual execution, particularly due to library encountered towards the semester's end.
- The challenge with libOpenCM3, although unresolved, leaves behind a rich documentation for further investigation.

- A flexible performance benchmarking system is in place, which enables future optimizations.
- SABER-KEM and FrodoKEM are not running, neither virtually nor on hardware. They used to work in the virtual environment, but after some updates we made, they stopped working, and we ran out of time to figure out why. However, with more effort, we believe they can be made functional again. The integration on the board is enabled by our framework, but requires implementations specifically for hardware.

4.3 Future Work

Extending Benchmarks. To provide a comprehensive understanding of memory usage and execution speed, it would be beneficial to measure and compare different hardware and system configurations, such as using 256bit-AES, 128bit-shake or the m4fstack implementation of pqm4-Kyber. Different KEMs should also be benchmarked, once implemented on the board. Extending the benchmarking to measure these configurations would provide a more detailed overview of memory usage patterns and potential areas for further optimization.

libOpenCM3 [lib] Library. The first step to addressing the issues we had with integrating libOpenCM3 [lib] would be looking at how the framework interacts with the STM32 HAL. Since we suspect there to be potential conflicts between the two, a further look into the USART communication and interrupt handling could lead to the source of the matter. Other than that, exploring and using more robust and better-equipped debugging tools would aid this process, especially with suspected problems in the interrupt methods.

SABER [DKRV18] and Frodo [BCD+16]. With the infrastructure already provided by means of the PAKEM core, running and benchmarking the PAKEM protocol using SABER and Frodo for the Key Encapsulation process would be the next step. The current official implementation of SABER depends on the OpenSSL library, which was difficult to compile for our STM32 board. Circumventing this obstacle could be achieved by finding a better-suited implementation of SABER or adapting the currently used implemtation to our needs. (Frodo...)

Initialization Vector for AES Encryption. AES encryption and decryption requires an initialization vector. The previous implementation had realized the Initialization Vector (IV) in a static, and hence not cryptographically secure matter. Due to time constraints and by instruction of our supervisor we left this part untouched, with the IV being "hardcoded" on both server and client side. Even though we have not made any changes in this regard, the acknowledgement message we implemented to set the mode of operation could be used to communicate the IV between client and server. In order to be considered cyptographically secure, the IV must be randomly generated and must be incremented after each encryption and decryption exchange.

Code Cleanup. Making the final transition from our virtual environment to running PAKEM in an actual server-client setting took quite a few changes to our code. Some fragments of these changes could still be cleaned up, improving the readability of the infrastructure.

Adapt Code for real-world deployment. The current state of the implementation is still a proof of concept. In order to prepare the PAKEM implementation for a real-world deployment, the code infrastructure still has to be expanded. Looking at what happens before and after the protocol execution itself and adding these steps to the code would be the next big step towards making the program more robust and ready to be used in practise.

References

BCD+16. Joppe Bos, Craig Costello, Leo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1006–1018, New York, NY, USA, 2016. Association for Computing Machinery.

BDK+18a. Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. Crystals - kyber: A cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367, 2018.

BDK+18b. Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

cod22. code.fbi.h-da.de, 2022. `https://code.fbi.h-da.de/aw/prj/athenepqc/mpse-eid-implementation/-/wikis/Basic-Cryptography`; abgerufen am 26.06.2023.

DKRV18. Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In *Progress in Cryptology – AFRICACRYPT 2018*, 2018.

hte. hterm. `https://gist.github.com/lalten/070870c6a2db4a14ff3a1c1a18996c25`.

LBB22. Chao Lu, Utsav Banerjee, and Kanad Basu. Design and analysis of a scalable and efficient quantum circuit for lwe matrix arithmetic. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 109–116, 2022.

lib. libopencm3. `https://github.com/libopencm3/libopencm3`.

NDGJ21. Kalle Ngo, Elena Dubrova, Qian Guo, and Thomas Johansson. A side-channel attack on a masked ind-cca secure saber kem implementation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 676–707, 2021.

RK19. Kumar Sekhar Roy and Hemanta Kumar Kalita. A survey on post-quantum cryptography for constrained devices. *International Journal of Applied Engineering Research*, 14(11):2608–2615, 2019.