M-PSE Project Report: Implementation of eID protocols – WS 22/23

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Date:	16.03.2023		
Students:	Ben Dlala	Ilyes	762960
	Geißner	Hans	1115172
	Kanellakopoulos	Jean	764931

1 Introduction

1.1 Key Information on the Project

Supervisor	Nouri Alnahawi
Time Period	Winter Semester 22/23
Initial State	A rectified version of the PACE protocol was already implemented (using C++). Nonetheless, there is a problem in the communication of the microcontroller with the computer, that is acting as the card terminal. In the virtual environment, however, the protocol runs as designed.
Goals	The project work basically had two main goals: First, the PACE protocol should fully run not only in the virtual environment but also on the microcontroller. In addition, the project was to be migrated from C++ to C. Optionally, the used cryptographic library as well as the one used for handling the microcontroller could be replaced with more suitable candidates. Also, the applied (PQC-safe) PACE protocol could be replaced with a slightly refined version. Later on, the need for time-benching capabilities on the microcontroller arose.
Sponsor(s)	German Federal Office for Information Security (BSI)

1.2 Motivation

With the advent of quantum computers, cryptographic methods that were previously considered secure are coming under attack as the new generation of computing devices can efficiently solve certain problems, such as prime factorization or the calculation of the n-th root in a residue ring. This entails a slew of changes, especially for information security, since cryptographic primitives such as RSA or

Diffie-Hellman rely on the complexity of the aforementioned mathematical problems. The consequence of this is that those cryptographic primitives need to be replaced by others that won't become endangered. In the context of eID cards, the PACE protocol stands out, since it uses a conventional Diffie-Hellman key exchange to establish a secure session with a card reading terminal. Diffie-Hellman is based on the complexity of the n-th root problem, so an alternative to this approach has to be found. The overall goal is to explore suitable frameworks for a post-quantum era by testing the improved PACE candidate under comparatively realistic conditions. This undertaking is supported by the project, which provides the refined PACE protocol.

1.3 Hardware

As mentioned above, the new PACE protocol is to be tested on a microcontroller simulating the eID card. The *NUCLEO-L4R5ZI-P* board was and will be used for this purpose within the scope of the project. With a suitable configuration, the technical capabilities of the board come close to those of an eID. The card reading terminal on the other side can be simulated with a conventional computer, since bottlenecks in the program flow are expected to arise in the eID, which is a lot weaker in terms of computing power.

2 Contributions

2.1 Revision: Rectified PACE Protocol

In this section, we will revisit the rectified PACE Protocol, i.e. Kyber-Ding protocol from the report of the previous semester. This is necessary for context for any further discussion. It is, as the name suggests, an improvement upon the PACE protocol that utilizes Kyber as a key encapsulation method as described by the paper by Ding et al. [1] that proposes a PAKE (Password Authenticated Key Exchange) based on RLWE (Ring learning with errors). It is a protocol designed to establish secure communication channels between an eID card and a terminal.

Below is an expanded description of the protocol's steps and an accompanying simplified graph of the protocol in Figure 1:

1. Phase: **Nonce Exchange**
    a) The eID card generates a random nonce, which serves as a unique, one-time-use value, ensuring that the same communication sequence is never repeated.

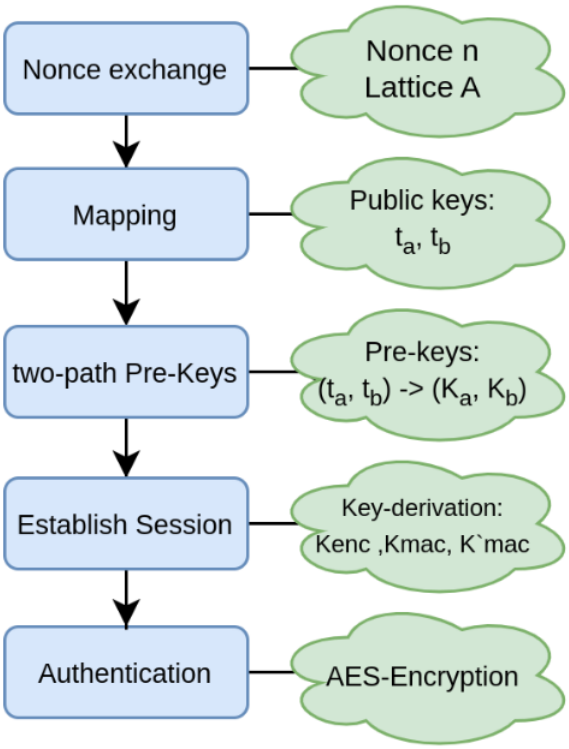

Figure 1: Rectified PACE Protocol: Kyber-Ding PACE

b) The eID card then encrypts the nonce using a key derived from the hashed PIN. This step protects the nonce from unauthorized access and ties the secure communication to the correct PIN.

c) The eID card generates the lattice base 'A', a crucial component in the Learning With Errors (LWE) cryptographic scheme, and sends it alongside the encrypted nonce to the terminal.

d) The terminal, using the PIN-derived key, decrypts the nonce, ensuring that only the terminal with the correct PIN can proceed with the protocol. The PIN is of course inputted by the user at the Terminal.

2. Phase: **Mapping**

a) Both the eID card and terminal independently compute their ephemeral public keys using the LWE cryptographic approach. This computation relies on the lattice base 'A' and other parameters to create a secure key exchange mechanism.

b) To add an extra layer of security, the eID card masks its public key with the hashed nonce, making it more difficult for an attacker to recover the original public key.

c) The terminal and the eID card exchange their public keys: the masked public key from the eID card and the regular public key from the terminal.

d) The terminal unmasks the eID card's public key using the decrypted nonce, recovering the original public key and validating the correctness of the PIN.

3. Phase: **Two-path pre-keys**

a) The terminal and the eID card encapsulate pre-keys using the exchanged ephemeral public keys. These pre-keys are essential for deriving the master key and establishing a secure communication channel.

b) Both parties exchange their encapsulated messages and decapsulate them to obtain pre-keys K_a and K_b.

4. Phase: **Authentication**

a) The terminal and the eID card derive a master key 'K' from the pre-keys K_a and K_b, which will be used to generate session keys for secure communication.

b) Session keys, such as the encryption key (K_{enc}) and message authentication code keys (K_{MAC}), are created using the master key 'K'. These keys ensure the confidentiality and integrity of the data exchanged between the eID card and the terminal.

c) Both parties generate authentication tokens (T_A and T_B), which contain one of the MAC keys, the lattice base 'A', and each other's public keys. These tokens verify that both parties have successfully generated the correct keys and have access to the secure communication channel.

5. Phase: **Establish Session**

a) Using the other MAC key and the encryption key (K_{enc}), the terminal and the eID card establish a secure session for encrypted communication, preventing eavesdroppers from intercepting or tampering with the exchanged data.

2.2 Programming Language Migration: C++ to C

One of the main contributions that we made is the migration of the project's codebase from C++ to C. This decision was encouraged by various benefits and resulted into an intensive migration process. We will take a look at both of these aspects in this section.

2.2.1 Reasons to use C Programming language

- Performance: C is a compiled language, which means that it is translated into native machine code before execution, allowing for optimal performance. This low-level control and direct access to hardware resources is crucial when implementing embedded applications, where efficiency and speed are of utmost importance. C's simplicity and lack of overhead also contribute to its faster execution times, making it more suitable for security applications compared to C++.

- Portability: In the realm of security, ensuring that cryptographic algorithms perform consistently across a diverse range of platforms is crucial. C is known for its portability, as it is widely supported by various operating systems, compilers, and hardware architectures. This feature enables developers to write cryptographic implementations once and deploy them on multiple platforms with minimal modifications, reducing the likelihood of platform-specific security vulnerabilities.

- Standardization: C has been an established standard in the programming world for decades, with the ANSI C and ISO C standards governing its specifications. This standardization simplifies the process of comparing, auditing, and reviewing cryptographic implementations, as developers can rely on a consistent set of rules and guidelines. In contrast, C++ has a more complex standard library and additional features that may introduce inconsistencies and make it harder to ensure the security of cryptographic implementations.

- Security: The simplicity of C's programming model, with its reduced feature set compared to C++, minimizes the number of ways developers can write incorrect or insecure code. This aspect helps reduce the risk of security vulnerabilities introduced by programming errors, such as buffer overflows, memory leaks, and uninitialized variables by making them more debatable. Moreover, C's low-level control over hardware resources, like direct memory access and manipulation, is crucial for implementing secure cryptographic algorithms that require precise control of memory and CPU operations.

In summary, C's performance, portability, standardization, and security make it a better choice for security applications and cryptographic implementations compared to C++. Its low-level control, simplicity, and widespread support contribute to its suitability for developing secure and efficient cryptographic algorithms that can be deployed across various platforms.

2.2.2 Migration Process

Migrating from C++ to C was a time-consuming task. But due to the advantages it brought, it was considered worth the effort. In this section, we will look into the various aspects that were changed during the migration.

- **Managing Classes and Objects:** C++ is an object-oriented language, while C isn't. The first major challenge when migrating a code base from C++ to C is converting classes and objects to equivalent structures and functions in C. For instance, consider the following C++ class:

```
class UartTransport : public Transport {
    private:
        int serial_port = -1;
        struct termios tty;
    public:
        UartTransport(const std::string& filename);
        void sendData(const std::vector<unsigned char>& data)
            override;
        std::vector<unsigned char> UartTransport::receiveData
           ()
    };
}
```

  This class can be converted to a C structure with associated functions:

```
struct UartTransport
{
    int serial_port;
    struct termios tty;
};

struct UartTransport* constructUartTransport(const char*
   filename);
void destructSocket_uart (struct UartTransport* ut);
void sendData_uart(struct UartTransport *ut, const struct
   ByteArray data);
struct ByteArray receiveData_uart(struct UartTransport *ut);
```

- **Handling Constructors and Destructors:** C++ constructors and destructors are not available in C, so we needed to use initialization and cleanup functions instead. The example above demonstrates the use of an initialization and destruct function for the *UartTransport* struct, i.e. *constructUartTransport()* and *destructSocket_uart()*.

- **Replacing C++ Standard Library Features:** C++ has an extensive standard library that provides many useful features, such as containers and algorithms. When migrating to C, we needed to find alternative implementations or libraries for these features.

  For instance, std::vector might be used in C++:

```
std::vector<unsigned char> paddedMessage;
...
paddedMessage.insert(paddedMessage.begin(), message.begin(),
   message.end());
```

  In C, you can replace it with a dynamic array:

```
typedef unsigned char byte;
...
byte* paddedMessage = (byte*) calloc(1, messageLen +
   AES_KEY_SIZE_BYTE - x);
memcpy(paddedMessage, message, messageLen);
```

- **Function Overloading:** C++ supports function overloading, allowing you to define multiple functions with the same name but different parameters. C does not support function overloading, so giving each function a unique name is needed. Having different input parameters is not enough. We chose to use a suffix to designate to which former class/struct the function belongs to, as seen in the following:

```
struct Nonce* constructNonce_client(const char* const pin);
struct Nonce* constructNonce_server(const char* const pin,
   const unsigned char* const z);
```

- **Managing Memory:** C++ provides constructors, destructors, and smart pointers for automatic memory management. In C, however, we must manage memory manually using functions like *malloc*, *calloc*, *realloc*, and *free*. Attention to memory leaks and dangling pointers when migrating from C++ to C is important. Tools like Valgrind help to detect these memory issues in most cases.

- **Type Casting:** C++ uses *static_cast*, *dynamic_cast*, *const_cast*, and *reinterpret_cast* for different casting scenarios. C only has one casting mechanism, the C-style cast.

  A possible casting scenario in C++ would be:

```
kdf(key.data(), reinterpret_cast<const uint8_t *>(pin.c_str
   ()), pin.length());
```

In C, you would do:

```
kdf(nonce->key, (const uchar*) pin, strlen(pin));
```

In conclusion, this section provides an analysis of the process of migrating a code base from C++ to C, highlighting the key differences between the two languages and offering practical solutions for various challenges.

2.3 Experimental Setup

To evaluate the protocol with respect to the limited hardware of an eID, a prototype with adjustable hardware will be deployed. The goal is to create a functional hardware prototype and to create benchmarks relating to runtime with different hardware limitations. In the preceding semester, various hardware options were considered, and eventually the NUCLEO-L4R5ZI development board was chosen. Although the first steps towards a working prototype have already been made, an error caused the communication to be one-sided, i.e. the communication between server and client failed, which meant that the protocol could not have been tested so far. One of the main goals of this semester was to fix the communication bug and subsequently obtain a working prototype.

2.3.1 Identifying the error source

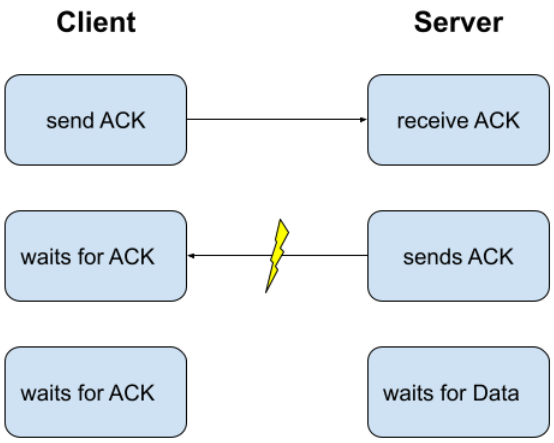

Figure 2: Communication Error

To start the protocol, the client sends an acknowledgment (ACK) to the server, which is then answered by the server by repeating the ACK. Figure 2 displays the communication between the client and the server: The client sends the ACK to the server,

which is received and then repeated, whereas the response never reaches the client. As a result, both the server and the client are waiting for each other.

To understand why the communication is failing, we have to take a closer look at the flow of the communication process.

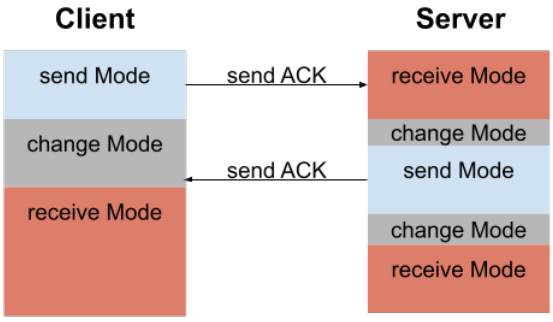

Figure 3: Communication flow

To receive a message, the receiving party has to be actively in receive-mode. If the receiving communication partner is not in receive mode, the message will be lost. In Figure 3 we can see that the communication error is caused by long transition times between send and receive mode on the client side. Those relatively long transition times are a direct consequence of the limited computing power.

2.3.2 Redesigning the communication flow

In order to fix the aforementioned bug, the communication flow is redesigned. Since the main problem in the communication was that the program had to be actively in receive mode, the rework focuses on implementing a routine that allows the client to be passively in receive mode, i.e. the receive mode is activated, but the client can do something else, e.g. sending a message, until an incoming message is detected. That is achieved by utilizing interrupts and a First-in First-out (FIFO) buffer.

The client activates the receive mode and every incoming message is stored into a FIFO buffer. When the next message is needed, the client can just retrieve it from the FIFO buffer.

2.4 Timing Capabilities

A functionality that was added this semester is the possibility to create time-benches on the microcontroller. This is especially important for the evaluation of the developed program and can further help to identify existing bottlenecks. The used board

has several timer/counter registers which can be used for measurement. In our case, a 64-bit counter register with an upstream prescaler is used. With an internal clock of 120 MHz and a prescaler factor of 60.000, this allows the capture of time with a precision of half a millisecond, whereas 2^{63} milliseconds ($\approx$ 292.5 million years) can be captured. In the context of a more realistic and thus lower clock in terms of our scenario, the timing precision would of course also decrease. But this can also be addressed by decreasing the prescaler factor. The following equation should be kept in mind.

$$Elapsed\ Time\ (in\ ms) = \frac{Prescaler\ Factor\ *\ 1000}{CLK}\ *\ Registered\ Ticks \tag{1}$$

Registered Ticks means the value that is stored and regularly incremented in the aforesaid timer/counter register. More detailed information on the use of the timing capabilities in the code can be found in the repository's wiki under *Experimental Setup on the Board*[1]. Note that it is the user's responsibility to obtain the registered ticks value from the microcontroller, e.g. by HAL communication. It is also the user's responsibility to transform the scalar tick count to a time span in seconds, according to Equation 1.

3 Conclusion

3.1 Review

In the beginning of the semester, we had 4 main goals:

- Refactoring the codebase to allow for better maintainability and code clarity
- Migrating from C++ to C
- Creating a timing benchmark system
- Upgrade Kyber-PACE scheme to version 2

The first three goals were fulfilled, yet the fourth one was not reached due to time limits.

3.2 Current-State

The project is in a stable state, with a codebase entirely in C and the protocol functioning in both virtual and hardware environments. The project now also includes a timebench, which enables further tests and benchmarks.

[1] `https://code.fbi.h-da.de/aw/prj/athenepqc/mpse-eid-implementation/-/wikis/Experimental-setup`

3.3 Outlook

Having successfully migrated the codebase to C and implemented necessary functions for benchmarking, future efforts should be focused towards measuring and improving performance. The next steps would be:

- Collect benchmarks with different hardware configurations
- Explore *libopencm3* as a possible replacement for HAL
- Upgrade Kyber-PACE scheme to version 2
- Replace primitives with the implementations contained in the *pqm4* library to improve efficiency

References

[1] Ding, Jintai & Alsayigh, Saed & Lancrenon, Jean & RV, Saraswathy & Snook, Michael. (2017). Provably Secure Password Authenticated Key Exchange Based on RLWE for the Post-Quantum World. 183-204. 10.1007/978-3-319-52153-4_11.