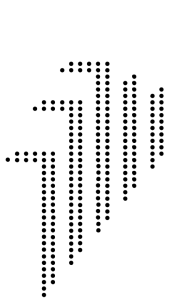

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Final Report MPSE: eID Implementation
MPSE WiSe 2021/22

Peter Müller[1], Matthias Merz[2], Dominik Heinz[3], Chiara-Marie Zok[4], Gero Knoblauch[5]

[1] Hochschule Darmstadt, Fachbereich Informatik, stptmue2@stud.h-da.de
[2] Hochschule Darmstadt, Fachbereich Informatik, stmamerz@stud.h-da.de
[3] Hochschule Darmstadt, Fachbereich Informatik, dominik.c.heinz@stud.h-da.de
[4] Hochschule Darmstadt, Fachbereich Informatik, stchzokk@stud.h-da.de
[5] Hochschule Darmstadt, Fachbereich Informatik, stgeknob@stud.h-da.de

Contents

1 Introduction

1.1 Project Description

Extended Access Control (EAC) [4] and Password Authenticated Connection Establishment (PACE) [5] are considered standard EU protocols for establishing secure communication between eCard chips and service terminals. Among other mechanisms, they are used to verify the terminal's access to data stored on the chip. These protocols are based on cryptographic schemes and primitives such as RSA [9] and DH [18], which will be classified as insecure according to NIST, due to the development of quantum computing [1].

The goal of the project is to implement and further develop these security protocols for the authentication of electronic documents (eID), such as the electronic identity card (ePA), and the electronic passport (ePassport), and to evaluate their capability and suitability for the expected migration to post-quantum cryptography (PQC).[11]

1.2 Goals

The main goal for the entire project is it to replace the conventional cryptography in the Password Authenticated Connection Establishment (PACE) protocol. This project was planned for a minimum of two semesters therefore we have the following sub goals for this semester:

- Understanding of the PACE Protocol
- Planning and implementing a proof of concept
- Research which Hardware we need
- Research which Post Quantum Cryptography (PQC) scheme we should take as replacement
- Optional: Finding a solution how to successfully replace the current cryptography in PACE

2 Theory

Work on theory-related topics mainly involved understanding important protocols such as PACE and finding suitable Post-Quantum Cryptography (PQC) algorithms. To be able to pick out a fitting PQC algorithm, constraints had to be considered. Those were:

- Fit on eID cards
- already have an available implementation
- Preferably NIST Round 3

2.1 Diffie-Hellmann

Diffie-Hellman is a widely used key exchange protocol, which enables secure key agreement over an insecure channel. It can either be implemented using discrete logarithms or elliptic curves. [6] Both implementations are threatened by Shor's algorithm, as soon as quantum computers become usable [7]. Therefore a suitable replacement to use in PACE protocol needs to be found.

2.2 PQC

To counteract the threat posed by Quantum Computers, the National Institute of Standards and Technology (NIST) has launched something like a competition to standardize PQCs [10]. This began in 2017 and is currently in Round 3, with the standards to be made available by 2024. The reason we focus on the NIST candidates is that these submissions already had a good code base, documentation, and benchmarks that simplified the decision process.
PQC schemes can be implemented in five different ways. These are either based on codes, lattices, hashes, mulitvariates or supersingular isogenies. Due to the previously defined constraints, it quickly became apparent that only lattice based schemes are suitable for our use case.
The following is a brief introduction to lattice-based cryptography. The general idea is to take a set of basis vectors and span an infinite lattice from them. Various problems can then be defined on this lattice. For example, the closest vector problem, where the closest lattice vector to a given point has to be found. In a 2D grid this is still easy to solve, but with higher dimensions it quickly becomes more complex. Most implementations are based on the Learning with Errors or Learning with Rounding problem.

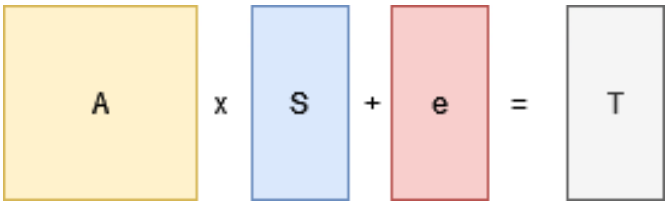

Fig. 1: Learning with Errors

As shown in Figure 1, a random uniform matrix A is multiplied by a secret vector s and a noise vector e is added. The matrix A and result T can then be used as public key and vector s as private key. The matrix and vectors are derived from a finite field or from a ring, when Ring LWE (RLWE) is used. A variation of LWE is LWR where instead of adding an error vector, deterministic rounding is used. Different implementations differ mainly in the underlying ring, the type of error sampling and the error correction. The underlying ring can be unstructured or structured (or ideal which is a subset of structured rings). The advantage of using a structured lattice are that key sizes are smaller and computations are faster. As a drawback structured lattices might not be as safe as unstructured lattices as their structure could be exploited. [2]

2.3 Kyber

For this project it became apparent that the smaller key sizes and faster computations are outweighing. From all the NIST candidates we had a closer look at NTRU, Kyber, Saber and ThreeBears. We decided to continue with Kyber [19] an indistinguishable Adaptive Chosen Ciphertext Attack (IND-CCA2) secure KEM over Module Lattices. Besides having the best documentation there is already an implementation available on constrained hardware. But what made Kyber special is that it offers three security levels, which are similar to those of AES-128/192/256. All security levels use the same underlying ring:

$$Z_q\,[x]\,/(x^n + 1)$$

This means that all calculations are implemented once and only the dimension k is changed to achieve a different security level. While most schemes use Gaussian sampling, Kyber uses centered binomial sampling (CBD), which further speeds up computations.

2.4 KEX, AKE and KEM

Keys can be established in different ways. In a Key Exchange (KEX) two parties establish a symmetric Key together. The Key Exchange can be complemented with an authentication mechanism, to prevent man in the middle attacks, which is then called an authenticated key

exchange. A different approach is using a Key Encapsulation Mechanism, where only one party establishes the key which is then encapsulated and sent to the other party. The PACE protocol uses an AKE but most NIST candidates are KEMs.

2.5 Extended Access Control (EAC)

EAC is a set of advanced security features which includes several protocols, Chip Authentication (CA) and Terminal Authentication (TA), for electronic passports that protects and restricts access to sensitive personal data contained in the RFID chip. It is executed along Basic Access Control (BAC), respectively PACE and Passive Authentication. A detailed description of these protocols can be found in the project wiki documentation.[15] The main goal of the MPSE is to replace all cryptography that is used in these protocols with post quantum cryptography. In the first semester the focus was set to PACE.[4]

2.6 Password Authenticated Connection Establishment (PACE)

PACE is executed before EAC to create a secure communication between terminal and chip, so they can exchange their certificates safely in TA and CA. The protocol includes several steps including a DH-Key agreement. PACE starts with generation of a random number called nonce, that is created by the chip instance. This nonce is encrypted by the PIN that is saved on each card. The nonce will be sent to the terminal and is decrypted by the user by typing in his PIN. After that the chip and the terminal exchange ephemeral domain parameters, which are created by a mapping protocol that is executed internally of PACE. Based on these ephemeral domain parameters terminal and chip perform an anonymous DH agreement. They derive their Key-Pairs from the ephemeral domain parameters and exchange their public key with each other. They generate a shared secret and derive session keys out of it. There is an Enc-Key, which is for the encryption for future communication and a MAC-Key, which can be used for authentication purposes. Now they encrypt the previously shared public keys with the MAC-Key, the chip uses the terminal-public-key and the terminal uses the chip-public-key. This is used as a token. They exchange this token and decrypt it with their MAC-Key to make sure both have the same keys. After that CA can begin. The DH-key agreement was the concerning part which had to be replaced, so we tried to replace it with a Kyber-PAKE. A detailed overview is shown by Figure 2.[5]

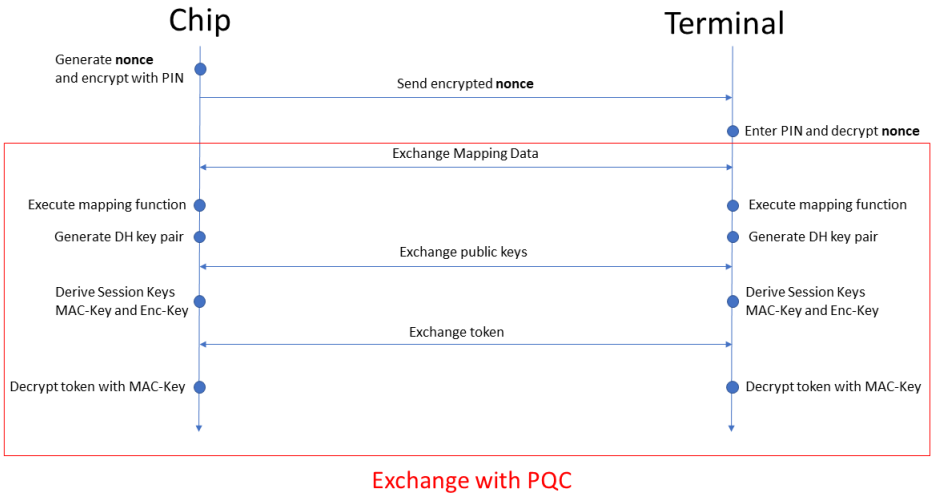

Fig. 2: PACE

2.7 Replacement problems

PACE has an attribute that should be handed over to the new protocol. The nonce has low entropy with an encryption of just 6 digits, so the knowledge of the nonce should not be essential to derive the keys. For that PACE uses the mapping protocol and the ephemeral domain parameters. This option doesn't exists in Kyber, so a new idea was sought. A temporary solution was implemented. The nonce was included in a hash value at the end of the Kyber-PAKE protocol (see Figure 3), but that is not secure against passive attacks. If someone would use a malicious terminal the right PIN could be cracked, by brute forcing till the right decrypted nonce is derived.

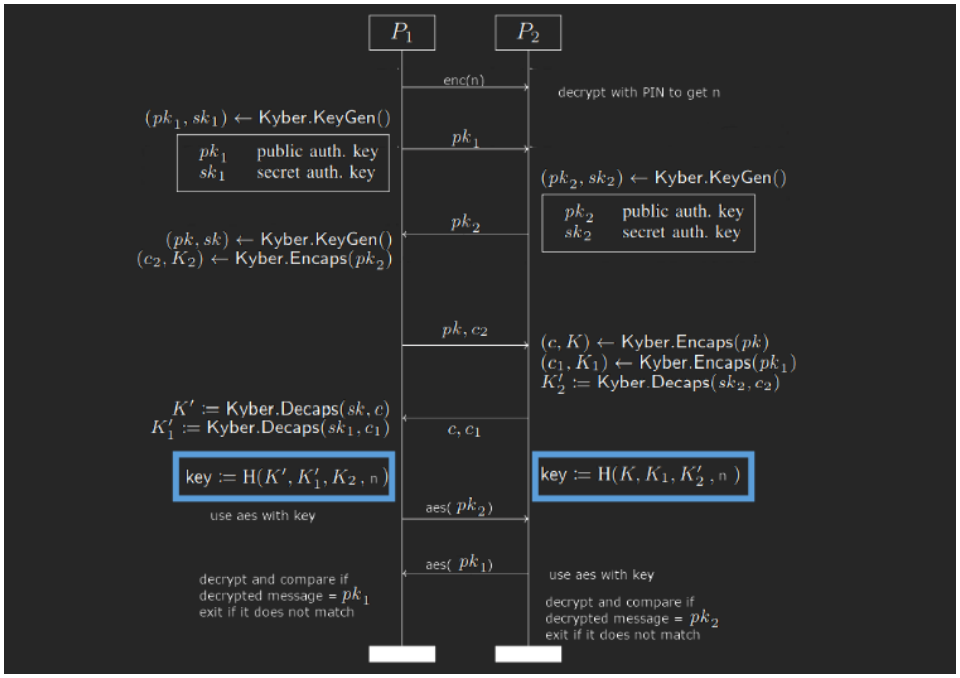

Fig. 3: Modified Kyber PAKE

3 Hardware

3.1 Limitations

Given the project focuses on the final implementation being on an eID or traveling document, we can assume that the current hardware restrictions apply for our use case.
If we take the current eID issued by the German Government the hardware specifications as shown in Table 1 apply. These specifications limit us in several aspects of our project: overall code size, size of keys, time taken to create keys and time taken to create ciphertext. Even if we assume that new chips will be available for next generation eID cards and their capacities in every compartment will be increased, the overall available computing power is limited by the fact that energy is supplied through NFC and the PQC implementations have not yet been realized in hardware.

Hardware	Specs
EEPROM	≥142.5 KB
USER ROM	≥586 KB
USER RAM	≥10176 B
CPU	8/16/24/32 bit instructions
CRC CoCPU	16/32 Bit

Tab. 1: Hardware specification fur currently issued personal IDs by the German Government, supplied by NXP as P60D145

3.2 Hardware for the Project

To realize the different scenarios, special hardware was acquired in order to reflect the restricted environment the eIDs outline. The decision which hardware should be acquired was based on several points:

1. Hardware should reflect the limitation of the target eID platform in a generalized manner, limitations in RAM, ROM and computational power should apply.
2. NFC capabilities should be included or in some way the hardware should be extendable to cover NFC capabilities.
3. If possible, implementations should be easily ported onto this new hardware platform.
4. Optional: Available optimizations in any form.

While taking these points into account we decided on several different development boards that covered the above mentioned points (As shown in Table 2).

Board	CPU freq,	Flash	RAM	Scenario
NUCLEO-L4R5ZI	120 MHz	2 MB	640 kB	PICC performance
NUCLEO-L476RG	80 MHz	1 MB	128 kB	reduced PICC performance
ST25R3911B-DISCO	80 MHz	512 kB	128 kB	NFC-Reader (PICC/PCD)
M24SR-DISCOVERY	72 MHz	1 MB	96 kB	NFC Tag (PICC)
ST25R3916-DISCO	80 MHz	1 MB	96 kB	Host Card Emulation

Tab. 2: Acquired hardware for testing PQC algorithms in a restricted environment to reflect eID limitations

The boards mentioned have been chosen as they cover our requirements as good as possible. Limitations in computation as well as on the storage side are given. Although they do not exactly match the hardware specifications of the NXP eID, we can impose even greater restrictions by ourselves and check if we are below the given boundaries of eID hardware. Furthermore there is already a PQC implementation for the *NUCLEO-L4R5ZI* development board, which allows us to utilize already tested and optimized code in our project.
Although we weren't able to test it during our project, two of the acquired five different models are capable of communicating via NFC. In the next period of this project, these boards can be used to test the implementations under the imposed limitations of the NFC interface.

During the project, the main focus and testing of software and implementations has been done on the *NUCLEO-L4R5ZI* - the code as well as the projects concerning the STM32 have been tested and evaluated on the aforementioned board.

3.3 Tools used for Hardware

The manufacturer offers tools for configuring, programming and flashing the development boards. Via the STM32CubeIDE [21] it is possible to program the device using the HAL-library, (an abstraction layer for ease of access) also the program can be live-debugged if a ST-Link is connected or directly on the development board, furthermore the output from the UART interface can be displayed with the included console window.
The cross-compilation for the target platform is done via the *gcc-arm-none-eabi* compiler that can be separately downloaded via Github [3] or through the official repository for Debian based distributions, if the STM32CubeIDE is not used. Also flashing the device is possible out-of-the-box via STM32CubeIDE, as an alternative solution the *st-flash* tool [20] can be used to flash and verify the binary onto the target platform.

3.4 Current Progress

Overall concerning the hardware, the project aims to: have OpenPACE and Crystals-Kyber run on the development boards and utilize the NFC interface provided by some of the

development boards.

As of the current state of the project we are able to include the pqm4 version [14] of Crystals-Kyber and have it successfully run on the *NUCLEO-L4R5ZI* development board [17]. Additionally our earlier attempts to build a library with crystals-kyber have resulted in a Makefile that allows us to build the library including all the components needed by Kyber and the development board.

Thus far, OpenPACE has not yet been successfully *ported* onto the development board, also the NFC functionalities have to be included into the project to properly reflect the use case of eID.

3.5 Open Issues

OpenPACE has to be included into the project, attempts to take the source files from the project on Github [12] and include them into the project have not been successful.
During our tests we encounter the following problems:

- OpenSSL: The project depends on OpenSSL - building the complete library produces a file that far exceeds the storage capabilities of the development board, added to this fact OpenSSL has building issues with the gcc-arm-none-eabi compiler and its source files provided by STM32CubeIDE.
- WolfSSL: To circumvent the issues with OpenSSL, WolfSSL was used as a substitute. WolfSSL provides a package for the STM32CubeIDE [22] that can easily be included into the project and allows for a successful build concerning the OpenSSL dependencies.
- OpenPACE: Through WolfSSL the building dependencies are covered, but the build process fails as several files in the source directory supplied by the gcc-arm-none-eabi compiler seem to have C formatting errors. The issue seems to be caused by the different data types, but this has not been verified as of this moment.

Kyber is currently running with all the source files supplied in the project, our intention was to have a library file so that there is no need to always check several projects and copy new files from the pqm4 project in Github into our own. At the moment the library can be built but throws errors if included and referenced inside the project:

- Kyber library: Our Makefile [16] is capable of building the library which allows us to include it into the project. When referenced and accessed, an error occurs referring to the *nttfast.S* file which is optimized assembly code (taken from pqm4) used to have increased computation speedup on the development board. When debugged, the code throws a *Hardfault* inside this section, which has not been resolved as of yet.

4 Implementation

4.1 Goals

For the implementation we had several goals:

- A working Testing lab for development.
- Implementing the PACE Protocol as a prototype.
- Find a solution to replace used cryptography. (optional)

4.2 Technology Stack

The following technology stack is used for prototype implementation:

- Docker
- OpenSSL
- OpenPACE [12]
- Kyber [13][8]
- C

all other dependencies that are needed for the Docker environment can be found in the file `/Sources/services/client/Dockerfile` and `/Sources/services/server/Dockerfile`. The dependencies for Kyber are already typed in both files but not tested.

4.3 Structure

Our structure simply reflects the two entities needed for the PACE protocol, a client (chip card) and a server (Terminal) (see Figure 4):

These two entities were developed with two approaches: A docker based implementation and a virtual machine based implementation. The virtual machine based approach was used as our main development environment, and the docker approach was worked on, to check if our implementation is reproducible on different Linux systems. This was mainly relevant, because the build process of dependencies like Kyber and PACE required system specific parameters. Both of our approaches of the prototype use the TCP protocol to communicate with each other (Exchange of parameters and data). TCP is meant as a placeholder for now,

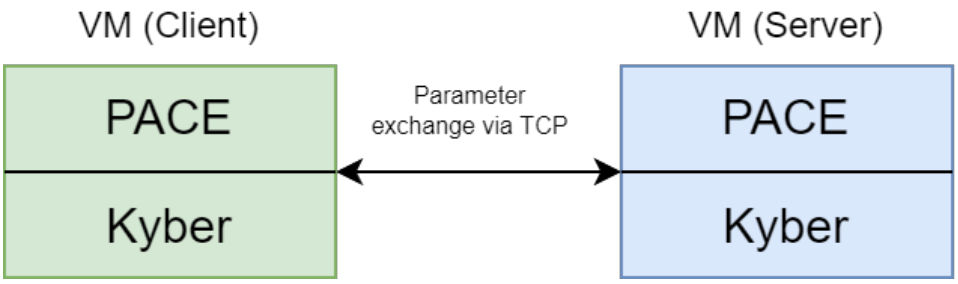

Fig. 4: Client and Server entities for Docker and VM approach

and should be replaced with other, contact-less protocols like NFC in future iterations. The implementation of our prototype was written in C11 and compiled with the clang compiler with no optimizations enabled.
The files for our docker prototype can be found under `/Sources/services` in our repository. The client and server implementation here both have their own Dockerfile. Under `/Sources/` we have several scripts for building the prototype and setting up a docker environment for the development. Docker Volumes are used to mount the source code and build scripts into the docker instance and build them there. For Building the application we recommend to use a IDE and also to use the already created VM for the development.

4.4 Current Status

At this point in time, in our implementation we successfully replaced Diffie-Hellman with a Kyber-PAKE. It is important to note that the VM-based implementation of our prototype was mainly used for development, and is more up to date than the Docker-based counterpart. We chose to work on the VM-implementation mainly because the development on the VM allowed us to use CLion as our IDE, which made debugging easier.

5 Outlook

5.1 Future Work

On the implementation side, there is a list of things that we couldn't finish, or were out of this semesters scope, but might be relevant in the future:

1. Update the Docker Environment: The Docker prototype is not on the same state as the VM prototype and should be updated.

2. Formal security proof: A formal security proof should be done, to verify that our modifications to the PACE protocol utilizing the nonce are secure
3. Expand our implementation to additionally include TA (Terminal Authentication) and CA (Chip Authentication).
4. Improve the implementation to accept new connections after success or failure. (Currently the server terminates after a successful exchange, and doesn't accept new client connections).
5. Refactoring the source code
6. Continue the implementation on the boards

References

[1] Lily Chen et al. *Report on Post-Quantum Cryptography*. URL: https://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf. (accessed: 28.02.2022).

[2] Prasanna Ravi et al. *Lattice-Based Key Sharing Schemes: A Survey*. URL: https://eprint.iacr.org/2020/1276.pdf.

[3] *arm-toolchain*. URL: https://github.com/marketplace/actions/arm-none-eabi-gcc-gnu-arm-embedded-toolchain. (accessed: 28.02.2022).

[4] *BSI-EAC*. URL: https://www.bsi.bund.de/EN/Topics/ElectrIDDocuments/SecurityMechanisms/securEAC/eac%5C_node.html. (accessed: 28.02.2022).

[5] *BSI-PACE*. URL: https://www.bsi.bund.de/EN/Service-Navi/Publications/TechnicalGuidelines/TR03110/BSITR03110.html. (accessed: 28.02.2022).

[6] *DHKE*. URL: https://cryptobook.nakov.com/key-exchange/diffie-hellman-key-exchange#security-of-the-dhke-protocol. (accessed: 28.02.2022).

[7] John Proos, Christof Zalka. *Shor's discrete logarithm quantum algorithm for elliptic curves*. URL: https://arxiv.org/abs/quant-ph/0301141.

[8] *modified pq-crystals kyber implementation*. URL: https://code.fbi.h-da.de/istmamerz/kyber-modifyed-for-pake. (accessed: 28.02.2022).

[9] Kathleen Moriarty et al. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Nov. 2016. DOI: 10.17487/RFC8017. URL: https://www.rfc-editor.org/info/rfc8017.

[10] NIST. *post-quantum-cryptography*. URL: https://csrc.nist.gov/projects/post-quantum-cryptography. (accessed: 28.02.2022).

[11] Alnahawi Nouri. *MPSE: Implementierung von eID Protokollen*. URL: https://fbi.h-da.de/personen/alnahawi-nouri/mpse-implementing-eid-protocols. (accessed: 28.02.2022).

[12] *OpenPACE*. URL: https://github.com/frankmorgner/openpace. (accessed: 28.02.2022).

[13] *pq-crystals kyber implementation*. URL: https://github.com/pq-crystals/kyber. (accessed: 28.02.2022).

[14] *pqm4*. URL: https://github.com/mupq/pqm4. (accessed: 28.02.2022).

[15] *Protocols*. URL: https://code.fbi.h-da.de/aw/prj/athenepqc/mpse-eid-implementation/-/wikis/Protocols. (accessed: 28.02.2022).

[16] *RepoMakefile*. URL: https://code.fbi.h-da.de/aw/prj/athenepqc/mpse-eid-implementation/-/blob/stm32/stm32/libkyber768speed/Makefile. (accessed: 28.02.2022).

[17] *RepotestPQClean*. URL: https://code.fbi.h-da.de/aw/prj/athenepqc/mpse-eid-implementation/-/tree/stm32/stm32/testPQClean. (accessed: 28.02.2022).

[18] Eric Rescorla. *Diffie-Hellman Key Agreement Method*. RFC 2631. June 1999. DOI: 10.17487/RFC2631. URL: https://www.rfc-editor.org/info/rfc2631.

[19] Roberto Avanzi et al. *CRYSTALS-Kyber, Algorithm Specifications And Supporting Documentation*. URL: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[20] *ST-Link*. URL: https://github.com/stlink-org/stlink. (accessed: 28.02.2022).

[21] *STM32CubeIDE*. URL: https://www.st.com/en/development-tools/stm32cubeide.html. (accessed: 28.02.2022).

[22] *WolfSSL*. URL: https://www.wolfssl.com/docs/stm32/. (accessed: 28.02.2022).