ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ

ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ

ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Δημιουργία ενός ολοκληρωμένου ασφαλή μικροελεκτή με χρήση hardware secure elements. Χρήση διάφορων αλγορίθμων για την σύγκριση ποιότητας. Δοκιμή νέων αλγορίθμων.

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Εισηγητής: Μπέρκη Γεωργία Μαρία, τπ4695

Επιβλέπων: Κορνάρος Γεώργιος, Καθηγητής

ΙΟΥΛΙΟΣ 2022

HELLENIC MEDITERRANEAN UNIVERSITY

SCHOOL OF ENGINEERING

DEPARTMENT OF INFORMATICS ENGINEERING

Creating an integrated secure microcontroller by using hardware secure elements. Using various algorithms for quality comparison. Testing of new algorithms.

DIPLOMA THESIS

Student: BERKI GEORGIA MARIA, TP4695

Supervisor: KORNAROS GEORGE, Associate Professor

July 2022

Abstract

The rapidly increasing field of lattice-based cryptography is one of the effective tools in quantum computing meant to replace the existing public-key systems. The mechanism FrodoKEM has been customized as an alternative encapsulation mechanism recently, which in fact is submitted to the post-quantum NIST process for standardization. In this setting, security is conditional on utilizing standard lattices and learning through tracing-correcting errors. In any case, the excessive number of parameters makes it difficult for the embedded systems to function properly. This approach based on the FrodoKEM scheme entails parameters allowing for its integration into smaller-sized devices through the means of basic lattice-based cryptography. This thesis proposes harnessing two low-cost microcontrollers, namely STM32L552 and STM32WL55JC1, fostering better performance for post-quantum cryptography on smaller-sized devices. For the needs of the implementation of the scheme, STM32L552 required two different implementations, one involving usage of 1-way cache and the other the usage of 2-way cache; as for STM32WL55JC1, due to the small storage capacity of the RAM, Flash was used to reduce RAM space. Thanks to the versatility of WLSSJC1 showcasing connectability to a LoRa Network, which is based on wireless technology, the findings of this process can prove valuable for an IoT system.

Εισαγωγή

Ο ταχέως αναπτυσσόμενος τομέας της κρυπτογραφίας με βάση το πλέγμα είναι ένα από τα αποτελεσματικά εργαλεία της κβαντικής πληροφορικής που προορίζεται να αντικαταστήσει τα υπάρχοντα συστήματα δημόσιου κλειδιού. Ο μηχανισμός FrodoKEM έχει προσαρμοστεί πρόσφατα ως εναλλακτικός μηχανισμός ενθυλάκωσης, ο οποίος μάλιστα υποβάλλεται στη μετα-κβαντική διαδικασία NIST για τυποποίηση. Σε αυτό το πλαίσιο, η ασφάλεια εξαρτάται από τη χρήση τυποποιημένων πλεγμάτων και τη μάθηση μέσω της ανίχνευσης-διόρθωσης σφαλμάτων. Σε κάθε περίπτωση, ο υπερβολικός αριθμός παραμέτρων δυσχεραίνει τη σωστή λειτουργία των ενσωματωμένων συστημάτων. Αυτή η προσέγγιση που βασίζεται στο σύστημα FrodoKEM περιλαμβάνει παραμέτρους που επιτρέπουν την ενσωμάτωσή του σε συσκευές μικρότερου μεγέθους μέσω της βασικής κρυπτογραφίας με βάση το πλέγμα. Η παρούσα διατριβή προτείνει την αξιοποίηση δύο μικροελεγκτών χαμηλού κόστους, συγκεκριμένα των STM32L552 και STM32WL55JC1, προωθώντας καλύτερες επιδόσεις για τη μετα-κβαντική κρυπτογραφία σε συσκευές μικρότερου μεγέθους. Για τις ανάγκες της υλοποίησης του σχήματος, ο STM32L552 απαιτούσε δύο διαφορετικούς ελέγχους, μια για 1-wayκρυφή μνήμη και μια για 2-way κρυφή μνήμη, ενώ όσον αφορά τον STM32WL55JC1 λόγω της μικρής αποθηκευτικής χωρητικότητας της SRAM, χρησιμοποιήθηκε η Flash μνήμη για τη μείωση του χώρου της SRAM. Χάρη στην ευελιξία του WLSSJC1 που επιδεικνύει συνδεσιμότητα σε ένα δίκτυο LoRa, το οποίο βασίζεται στην ασύρματη τεχνολογία, τα ευρήματα αυτής της διαδικασίας μπορούν να αποδειχθούν πολύτιμα για ένα σύστημα διαχείρισης δεδομένων IoT.

Contents

Acronyms and Abbreviations

AES	Advanced Encryption Standard
ART	Adaptive real time memory accelerator
CBC	Cipher Block Chaining
CCM	Cipher block chaining message authentication code
CPU	Central Processing Unit
CTR	Counter
CTS	Clear To Send
DMA	Direct Memory Access
ECB	Electronic Code Book
FIPS	Federal Information Processing Standards
GCM	Galois Counter Mode
GMAC	Galois Message Authentication Code
GPIO	General Purpose Input Output
GPU	Graphic Processing Unit
ICACHE	Instruction Cache
IoT	Internet of Things
IND-CCA	Indistinguishability under Adaptive Chosen-Ciphertext Attacks
IND-CPA	Indistinguishability underChosen Plaintext Attack
KEM	Key Encapsulation Mechanism
LBC	Lattice-Based Cryptography
LIN	Local Interconnect Network
LPUART	Low-power Universal Asynchronous Receiver Transmitter
LWE	Learn with Errors
NIST	National Institute of Standards and Technology
PRNG	Pseudo Random Number Generator
PWM	Pulse Width Modulation
RAM	Random Access Memory
RNG	True Random Number Generator
RSA	Rivest Shalmir Adleman
RTS	Request To Send
SHA3	Secure Hash Algorithm
SIVP	Shortest Independent Vector Problem
SRAM	Static Random Access Memory
SW	Software
SYS	System File
TIM-2	TIMER 2
USART	Universal Asynchronous Receiver Transmitter

Figures

Tables

1. Introduction

Post-quantum cryptography involves the use of algorithms intending to provide security in the event of a cryptanalytic attack given that quantum computing can be threatening as it makes use of such means which might be able to break the vast majority of the existing cryptographic systems. For the most part, the effectiveness of lattice-based cryptography relies on the fact that it paves the way to the implementation of advanced security guarantees, making it possible to replace the RSA- and discrete-based logarithm which is in use to date.

Lattice-based cryptographic modules follow a tracing-correcting errors system, known as the Learning With Errors problem (LWE), revealing both a worst-case and average-case reduction from the Shortest Independent Vector Problem (SIVP) [**Error! Reference source not found.**].LWE lays the foundation to producing numerous cryptographic algorithms as well as generating indistinguishability under chosen plaintext attack (IND-CPA) and indistinguishability under adaptive chosen-ciphertext attacks (IND-CCA) security guarantees [2].Later on, LWE was modified to more advanced and effective versions or variants, Ring-LWE and Module-LWE among others, which exploit more suitable module lattices and lattices accordingly[**Error! Reference source not found.**]; without necessarily strengthening the system's security or the system may be more vulnerable to attacks due to the bi-formatting of the algebraic composition.

In the Intelligent Systems and Computer Architecture Lab, which belongs to the department of Electrical and Computer Engineering of the Hellenic Mediterranean University, there has been a variety of studies examining implementations and developments due to the necessity of IoT and automotive security. [**Error! Reference source not found.**] accentuate the need for the adoption of layered systematic approach in terms of hardening the electronic architecture of vehicles against prospective cyber-attacks, unauthorized access and increase safety. To achieve greater safety changes ought to take place as regards the actual implementation on an electric vehicle whose infrastructure is based on secure interconnection tools, hardware firewall excluding interference and unauthorized access, and flexibility in individual OS instances for various execution environments aiming at providing support and deployment for applications, without being altered to the automotive platform. Among others, for an in-depth understanding of these issues, relevant works in this field offering an innovative perspective include research in hardware support for cost-effective system-level protection in multi-core socs[12], automotive virtual in-sensor analytics for securing vehicular communication [**Error! Reference source not found.**],and secure asset tracking in manufacturing through employing IOTA distributed ledger technology [14].

Without a doubt, there has been a determining impact of advanced digitalization technologies on the automotive industry, involving developments and drastic changes

in electric mobility, automation, autonomy and connectivity. Notwithstanding these developments, the increase in connectivity has also seriously increased the level of vulnerability as regards the growing number of attacks on vehicles. Hence, as automotive products become more and more automated, the need for upgrading security grows to a greater extent. The long timespan of products in the automotive industry showcase makes it necessary to take into consideration the risks currently existing along with the dangers that are likely to emerge in the future when designing automotive security [**Error! Reference source not found.**]

The ever-increasing improvements in quantum computing cryptography pose an actual risk for the security and sustainability in the automotive industry, signifying that Post-Quantum Cryptography (PQC) should be thoroughly integrated. Using lattice-basedPQC offers the opportunity for the development of hands-on and optimized implementations. Using lattice-based encapsulation mechanisms with integers, integrating an Error Correcting Code (ECC) allows for high error-correcting capability therefore increasing security and speed and, at the same time, decreasing the rate of failure so as to implement CCA transformation and avoid repeating the protocol [8],[23].

For this reason, introducing a scheme including FrodoKEM with the standard LWE is considered meaningful in terms of increasing the level and extent of security guarantees as its structure is comparatively less prone or vulnerable to algebraic threats.Minimizing communication bandwidth with regard to protocol can greatly assist in assuring stable performance in widely used functions across a certain range of devicesin which standardizing a KEM can help dealing with unprecedented cryptanalytic attacks with diverse structures against lattices. Such an upgrade in security so as to combat commonly popular threats can efficiently contribute to FrodoKEM dealing with security issues rooted in prospective cryptanalytic attacks in the long run.

There has not been enough evidence on the functionality of Frodo variants on embedded systems. In this vein, the overarching aim is to come up with practical solutions to create a balance between the gap of hands-on assessments of standard lattice-based cryptography and the demand for long lasting security strategies in connection with the Internet of Things, taking into account the large number of the conservative parameters involved in the overall design of Frodo variants. The fact that in embedded devices, like the microcontroller STM32WL55JC1, there is little memory, there needs to be particular emphasis placed on decreasing the rate of memory consumption in the implementation stages and, make sure that the computing functions of the platforms in use do not malfunction or underperform. On the other hand, the embedded system of STM32L552 has enough space in memory and needs less time to implement FrodoKEM.

2. Related works

This unit of related works is separated into three categories. The first category refers to attacks on FrodoKEM. The second one refers to the Internet of things (IoT) and the other one is the hardware and software updates.

2.1 Attack on FrodoKEM

The algorithm of FrodoKEM is such that it can deal with a variety of attacks. In this section, three papers [22], [11] and the [4] there are described three different kinds of attacks. The first kind is the side channel attack, the second one is the secret key-recovery known as timing attack, and the last is the single-trace attack methodology.

The method of detecting the side-channel attack is as follows, FrodoKEM implements two functions: Encryption and Decryption. During the process of Encryption, a part of the public key creates a ciphertext which then is separated in two parts, c1 and c2, where this function generates a share_secret. The main task taking place in the Decryption stage is unpacking c1 and c2 to generate these two parts with the usage of the secret key. If the generated parts are equal to c1 and c2, the share_secret, which is created, is the same as the share_secret of the Encryption. On the other hand, if c1 and c2 are not the same as the generated arrays, the share_secret, which is produced, is not similar to the share_secret of the Encryption. The share_secret is the information which indicates if the ciphertext has been attacked.[22]

Another kind of attack is the timing attack, when one tries to recover the secret key. Firstly, it is important to generate a valid ciphertext. More specifically, the ciphertext has to be properly decrypted. Secondly, the attacker has to find the matrix E''' which denotes the noise matrix and data so as to find out if they are known values. As a result, there are linear equations in the secret key value S if one can figure out the arrayE''. Furthermore, the attacker has to transfer the noise in different cases along with the public key and run some tests [11]. In the Chapter 4.2 Encaps or Encryption there are explanations of how the E''', E'' and S are implemented.

Another technique to attack the Lattice-Based Cryptography (LBC) is the single-trace attack. During message encoding, FrodoKEM scans two sensitive bits at a time. As a result, there are four instances of the extracted sensitive bit *w*value, namely the *w*values are (00)2, (01)2, (10)2, (11)2. Consequently, if *w*value equals (00)2, while extracting or saving the *w*value, power consumption linked to 0 occurs. If *w*value is more than zero, power consumption is proportional to Hamming weight. As a result, message m= (m-'1,..,m1, m0)2 may be extracted and a secret share_key K can be generated. The results of the studies [4] showed that the message *m= (m'-1,..., m1, m0)2* could be recovered with just a single trace. In the case of analyzing Hamming weight value of *w*value, the success rate was over 90.57%. Accordingly, the secret share_key K could be recovered by applying an exhaustive search of candidates.[4]

2.2 IoT system

Nowadays, the need for security in the Internet of things (IoT) has emerged. In an IoT system, different kinds of devices are connected in an attempt to significantly upgrade the security techniques which can protect the system itself against quantum computing through the post-quantum algorithm. To be more precise, FrodoKEM is a post-quantum algorithm which due to the existing parameters can be identified using a flawless post-quantum cryptosystem, such as pseudorandom generators, pseudorandom functions, and digital signatures.

In this thesis, four different options are presented aiming to use this post-quantum algorithm in an IoT system. The first one [9] describes how vulnerable an IoT system is and why it is necessary to use extensive keys and an algorithm such as FrodoKEM. The second one [**Error! Reference source not found.**] refers to two types of attacks, the side-channel attack and the timing attack with a similar type of algorithm to LBC. The two last papers [3],[11] present the basic reasons why it is necessary to use a post-quantum algorithm in such settings.

The first kind of attack is developed into an IoT system. This IoT system has to confirm a specific password every time. The developers [9] paper suggest upgrading the security of this system with the usage of multiple keys each time the system has to ask for another key and to confirm it. The last scenario is using a post-quantum algorithm, FrodoKEM, because it is made up with a huge key, and it can upgrade security with its technique. All of this is feasible due to a combination of a powerful pseudo-random function.

According to the second paper [**Error! Reference source not found.**], some attacks and threats, such as side-channel, depend on how the system is implemented. The fundamental purpose of side-channel assaults is to trace the relationship between physical design parameters like power consumption and timing behavior in order to exploit the secret key. The proposed design is secure against timing attacks for three reasons: (a) there are no conditional branches or dependencies between the inputs and the cipher-text, (b) the proposed design executes a constant number of clock cycles for decryption for each cipher-text, and (c) the proposed design for the critical path delay is constant in all three phases.

The characteristics which provide an LBC and FrodoKEM by extension are specific. Firstly, this technique is based on NP-hard issues with a range of hardness from medium to extreme. Secondly, in addition to bearing stable quantum age, LBC implementations are notable for their efficiency, owing to their inherent linear algebra-based matrix/vector operations on integers.The latest is custom security, which according to LBC has developed techniques such as identity-based encryption [26], attribute-based encryption [26] and fully homomorphic encryption [27], in

addition to the basic classical cryptographic primitives, such as encryption, signatures and key exchange solutions required in the quantum era.

2.3 Hardware and software update of FrodoKEM

There are studies [**Error! Reference source not found.**], [8], [19] which present new options to update the techniques of FrodoKEM and how to make the algorithm quicker. Most of them use the hardware accelerator of AES on their chips. These different tasks are described in the following paragraphs.

The researchers conducted a study centered on the Nvidia GPU. FrodoKEM requires many bytes each time, including many operations, namely, among others, randombytes, SHAKE, and AES. The function of AES requires the majority of the run time of the algorithm; this is the reason why the researchers performed the AES operation in parallel to using a GPU. [**Error! Reference source not found.**]

James Howe. Tobias Oder, Markus Krausz, Tim Güneyrecommended a different solution[8]utilizing the functions of SHAKE128 and AES128. Researchers used the AES optimized implementation proposed by Schwabe and Stoffen and an assembly implementation for the cSHAKE [5]. This implementation considerably reduced the time of running and the memory space.

The study of Bos, Friedberger, Martinoli, Oswald, and Stam [19] is about a new technique whoseperformance is about of matrix multiplications involving A. They generate the matrix A not with the two traditional forms of FrodoKEM which are the AES128 and cSHAKE128 but they change it with a new form which called PRNG xoshiro 128. It is a noncryptographic technique which is based on the PseudoRandom number generator and they suggest it as over-conservative for the process to speed up the generating for a public seed.

3. Algorithms

In this chapter the basic algorithms used by FrodoKEM are mentioned. These algorithms are SHAKE128 and AES128. Also described is the model on which the algorithm is based, which is Learn with Errors (LWE).

3.1 FrodoKEM

The FrodoKEM family introduced the key-encapsulation mechanisms (KEMs). FrodoKEM schemes are intended to be conservative yet workable post-quantum constructions whose security stems from careful parameterizations of the Learning With Errors System(LWE). Three different security levels are designed for IND-CCA: FrodoKEM-640, FrodoKEM-976, and FrodoKEM-1344. Except that, there are two variants for each level, AES and SHAKE128. AES uses AES128 to

pseudorandomly generate a large public matrix and SHAKE128 uses SHAKE128 to pseudorandomly generate the matrix.

FrodoKEM is based on FrodoPKE which is a public-key encryption technique whose IND-CPA security is closely linked to the complexity of a similar to LWE learning with errors problem. At first, Lindner-Peikert [18, 19] proposed a more efficient LWE-based public-key encryption technique that uses a square public matrix Arather than an oblong rectangular one. FrodoPKE system involves a modified version of the Lindner–Peikert scheme [28, 29]. FrodoPKE create a pseudorandom creation of the public matrix A from a tiny seed that has as results more balanced key and ciphertext sizes, and additional LWE settings [17].

3.2 SHAKE128

The algorithm SHAKE128 has been generated by SHA-3, which is a hash function entails several implementations all of which are specified as an instance of the KECCAK-p family of permutations in this Standard to allow for the modification of its size and security parameters [6].One of them is an extendable-output function (XOF), which is a bit string function whose output can be extended to any length. The difference between SHA-3 and SHAKE is that SHA3 is a function on binary data for which the length of the output is preset. For instance, if the name of SHA3 changes to SHA3-256 the output demonstrates 256bits. Instead, the logic behind SHAKE is that the suffixes "128" and "256" indicate the security strengths that these two functions may commonly make use of.

The logic of the SHAKE algorithm operation involves an input in the hash function is called *message*, and the output *message* is called *digest* or *hashvalue*, providing the desirable length. The reason for using SHAKE128 is that it is a hash function which is used in a wide range of information security applications, including digital signature production and verification, key derivation, and the generation of pseudorandom bits [6].

3.3 AES

The standard of AES-128 provides the Rijndael algorithm, a symmetric block cipher capable of processing 128-bit data blocks with 128-bit cipher keys. The AES algorithm uses 128-bit sequences and digits with values of 0 or 1 for both input and output. For the AES algorithm, the Cipher Key is a 128-bit sequence. The bits in such sequences are numbered from zero to one less than the length of the sequence (block length or key length). Because the length and key length of the block are both 128nit, the number I attached to a bit is known as its index, and it is going to be in one of the ranges: 0 £ I 128 [1].

The basic logic of the algorithm is based on two functions: the encryption and Decryption. Firstly, to encrypt a text there needs to be as a parameter the key of 128bits and the text which has to be divided with 128 bits. As an output, there is an array of the ciphertext, which has the same length of the text. The second step is to decrypt the ciphertext with the correct 128-bit key which has to produce the text [1].

3.4 Learning With Errors

The LWE problem refers 'the problem of decoding random linear codes' [18, p. 3]. The LWE is constructed around three functions, to be precise. The first function creates public and private keys, whereas the second and third functions are encryption and decryption [**Error! Reference source not found.**].

Private key: The private key is a vector s uniformly chosen from Z_q^n.

Public Key: The public key consists of m samples $(a_i, b_i)_{i=1}^m$ from LWE distribution with secret s, modulus q, and error parameter α.

Encryption: For each bit of the message, do the following. Choose a random set S uniformly among all 2^m subsets of [m]. The encryption is$(\sum_{i\in s} a_i, \sum_{i\in s} b_i)$ if the bit is 0 and $(\sum_{i\in s} a_i, \frac{q}{2} + \sum_{i\in s} b_i)$ if the bit is 1.

Decryption: The Decryption of a pair (a, b) is 0 if b−$\langle a, s\rangle$ is closer to 0 than$\frac{q}{2}$ to modulo q, and 1 otherwise.

There are two types of LWE. The first is the search problem, in which the secret s is recovered from a set of samples drawn from the LWE distribution, and the second is the prediction problem, in which the secret s∈Z_q^n is predicted from a set of samples drawn from the LWE distribution. In the decision stage, a set of samples obtained from the LWE distribution from uniformly picked random samples is identified. The uniform distribution and the (nmodq) distribution, in which each coordinate is picked from the error distribution and reduced modulo q, are widely studied for both forms of the secret s∈Z_q^n. The latter is commonly known as the "normal form"of LWE [**Error! Reference source not found.**].

4. FrodoKEM Structure

The algorithm has three basic functions: the KeyGen(), the Encaps() and the Decaps(). The KeyGen() is the function which generates the two keys: the first key is the public key and the second one is the secret key. Encaps() uses the public key as parameter and creates two arrays the ciphertext and the shared_secret. Decaps() uses the secret key and ciphertext as parameters; with the processing of these functions, another share_secret is generated for Decaps(). If the two share_secrets are equal that means that there is not a kind of attack, if they are not similar there is an attack. In each function there are other functions which include basic cryptographic systems, such as AES and SHAKE128 and some techniques which create Gaussian noise and others [2].

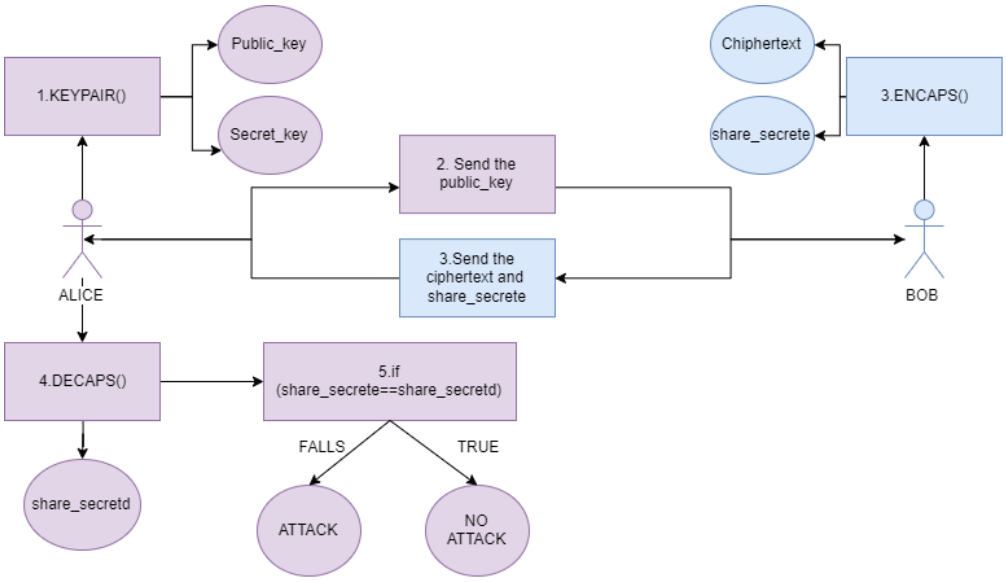


Figure 1: Alice and Bob implement step by step the algorithm of FrodoKEM

4.1 Keygen or Keypair

To begin with, KeyGen() refers to certain parts of code and its function is under the name Keypair [2].The function has the public key and the secret key as parameters. First, the public key is created and after, with the usage of this key, the secret key is generated.

Figure 2 and Figure 3 represent the implementation of the KeyGen(). The implementation begins with the creation of a random matrix with the name seedA and consists of 16bits; then with the help of those 16 bits the function Frodo.Gen (i.e.,

seedA) generates the matrix A. Frodo.Gen or seedA has two forms. The first one uses the crypto algorithm SHAKE128 and the other one the AES; it is up to the user to decide on what kind of technique they want to use. Moreover, a pseudorandom bit string is generated, and that information creates two arrays, the array S and the array E, with Frodo.SampleMatrix. The next step is to compute the arrays A, S and E with the combination B = AS+E, with B being equivalent to Frodo.Pack(B). Furthermore, it is important to create the arraypkh with the computation pkh=SHAKE(seedA||b,len(pkh)). The last step is to return the public key or pk=(seedA||b) and the secret key or sk=(s||seedA||b,S,pkh).

Input: None.
Output: Key pair (pk, sk') with $pk \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_\mathbf{A}}+D\cdot n\cdot\overline{n}}$, $sk' \in \{0,1\}^{\mathsf{len}_\mathbf{s}+\mathsf{len}_{\mathsf{seed}_\mathbf{A}}+D\cdot n\cdot\overline{n}} \times \mathbb{Z}_q^{\overline{n}\times n} \times \{0,1\}^{\mathsf{len}_{\mathbf{pk}}}$

1: Choose uniformly random seeds $\mathbf{s}\|\mathsf{seed}_{\mathbf{SE}}\|\mathbf{z} \leftarrow_{\$} U(\{0,1\}^{\mathsf{len}_\mathbf{s}+\mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}}+\mathsf{len}_\mathbf{z}})$
2: Generate pseudorandom seed $\mathsf{seed}_\mathbf{A} \leftarrow \mathrm{SHAKE}(\mathbf{z}, \mathsf{len}_{\mathsf{seed}_\mathbf{A}})$
3: Generate the matrix $\mathbf{A} \in \mathbb{Z}_q^{n\times n}$ via $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_\mathbf{A})$
4: Generate pseudorandom bit string $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2n\overline{n}-1)}) \leftarrow \mathrm{SHAKE}(\mathsf{0x5F}\|\mathsf{seed}_{\mathbf{SE}}, 2n\overline{n}\cdot\mathsf{len}_\chi)$
5: Sample error matrix $\mathbf{S}^{\mathrm{T}} \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(n\overline{n}-1)}), \overline{n}, n, T_\chi)$
6: Sample error matrix $\mathbf{E} \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(n\overline{n})}, \mathbf{r}^{(n\overline{n}+1)}, \ldots, \mathbf{r}^{(2n\overline{n}-1)}), n, \overline{n}, T_\chi)$
7: Compute $\mathbf{B} \leftarrow \mathbf{AS} + \mathbf{E}$
8: Compute $\mathbf{b} \leftarrow \mathsf{Frodo.Pack}(\mathbf{B})$
9: Compute $\mathbf{pkh} \leftarrow \mathrm{SHAKE}(\mathsf{seed}_\mathbf{A}\|\mathbf{b}, \mathsf{len}_{\mathbf{pkh}})$
10: **return** public key $pk \leftarrow \mathsf{seed}_\mathbf{A}\|\mathbf{b}$ and secret key $sk' \leftarrow (\mathbf{s}\|\mathsf{seed}_\mathbf{A}\|\mathbf{b}, \mathbf{S}^{\mathrm{T}}, \mathbf{pkh})$

Figure 2: Algorithm FrodoKEM.KeyGen.(as appended to Alkim et al., 2021, p. 21)

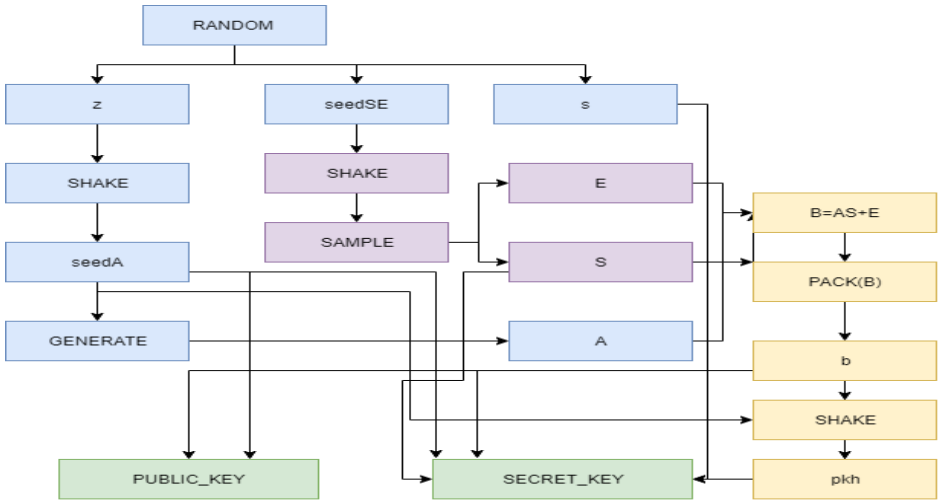


Figure 3: Public and Secret Key creation step by step.

4.2 Encaps or Encryption

Encaps() or Encryption is required to generate the cipher text and the first share_secret. The Encaps function uses the public key to generate the cipher text and the share_secret as parameters and is named

Encaps(publickey,ciphertext,share_secret). The public key is shared by the user of the algorithm because they ought to evaluate the reason for the improvement in the safety of the community at each time [2].

Figure 4 and Figure 5 present the stages in which Encaps() is implemented. Implementation begins with the creation of a key matrix with the name m. In the space which is similar to both public key and secret key, SHAKE128 and with this pkh is computed and created. The next step is to generate two pseudorandom arrays with the Frodo function.SHAKE to create the seedse and a random bit of string. This random bit of string is separated in three parts and each part is processed with the Frodo.SampleMatrix which normalizes the arrays; more specifically, it samples the error matrix and creates the three matrices: E', S' and E''. Moreover, the system generates a new array which is called A. This implementation includes giving the first 16 bit of the public key to the Frodo.Gen(seedA) function. After that the next step is to compute the arrays A, S' and E' with the combination B'=AS' + E' with the B' producing c1=Frodo.Pack(B'). Furthermore, the other step is to unpack b which is a part of the public key and create the B, B=Frodo.Unpack(b). The B,S' and E'' are computed and produce the V=S'B+E'' and the V is used to produce the array C, C=V+Frodo.Encode(m) . C is implemented to create c2 with the function pack, such as the c2=Frodo.Pack(C). The last step is to compute the share_secret which is named as ss, and the computation is implemented with the function SHAKE128 where ss=SHAKE(c1||c2||k,lenss).

Input: Public key $pk = \mathsf{seed}_{\mathbf{A}} \| \mathbf{b} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}} + D \cdot n \cdot \overline{n}}$.
Output: Ciphertext $\mathbf{c}_1 \| \mathbf{c}_2 \in \{0,1\}^{(\overline{m} \cdot n + \overline{m} \cdot \overline{n})D}$ and shared secret $\mathbf{ss} \in \{0,1\}^{\mathsf{len}_{\mathsf{ss}}}$.

1: Choose a uniformly random key $\mu \leftarrow_{\$} U(\{0,1\}^{\mathsf{len}_{\mu}})$
2: Compute $\mathbf{pkh} \leftarrow \mathrm{SHAKE}(pk, \mathsf{len}_{\mathbf{pkh}})$
3: Generate pseudorandom values $\mathsf{seed}_{\mathbf{SE}} \| \mathbf{k} \leftarrow \mathrm{SHAKE}(\mathbf{pkh} \| \mu, \mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}} + \mathsf{len}_{\mathbf{k}})$
4: Generate pseudorandom bit string $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2\overline{m}n + \overline{m}\overline{n}-1)}) \leftarrow \mathrm{SHAKE}(\mathtt{0x96} \| \mathsf{seed}_{\mathbf{SE}}, (2\overline{m}n + \overline{m}\overline{n}) \cdot \mathsf{len}_{\chi})$
5: Sample error matrix $\mathbf{S}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\overline{m}n-1)}), \overline{m}, n, T_{\chi})$
6: Sample error matrix $\mathbf{E}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(\overline{m}n)}, \mathbf{r}^{(\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n-1)}), \overline{m}, n, T_{\chi})$
7: Generate $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
8: Compute $\mathbf{B}' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
9: Compute $\mathbf{c}_1 \leftarrow \mathsf{Frodo.Pack}(\mathbf{B}')$
10: Sample error matrix $\mathbf{E}'' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(2\overline{m}n)}, \mathbf{r}^{(2\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{m}\overline{n}-1)}), \overline{m}, \overline{n}, T_{\chi})$
11: Compute $\mathbf{B} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{b}, n, \overline{n})$
12: Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
13: Compute $\mathbf{C} \leftarrow \mathbf{V} + \mathsf{Frodo.Encode}(\mu)$
14: Compute $\mathbf{c}_2 \leftarrow \mathsf{Frodo.Pack}(\mathbf{C})$
15: Compute $\mathbf{ss} \leftarrow \mathrm{SHAKE}(\mathbf{c}_1 \| \mathbf{c}_2 \| \mathbf{k}, \mathsf{len}_{\mathbf{ss}})$
16: **return** ciphertext $\mathbf{c}_1 \| \mathbf{c}_2$ and shared secret $\mathbf{ss}$

Figure 4: Algorithm FrodoKEM.Encaps. (as appended to Alkim et al., 2021, p. 21)

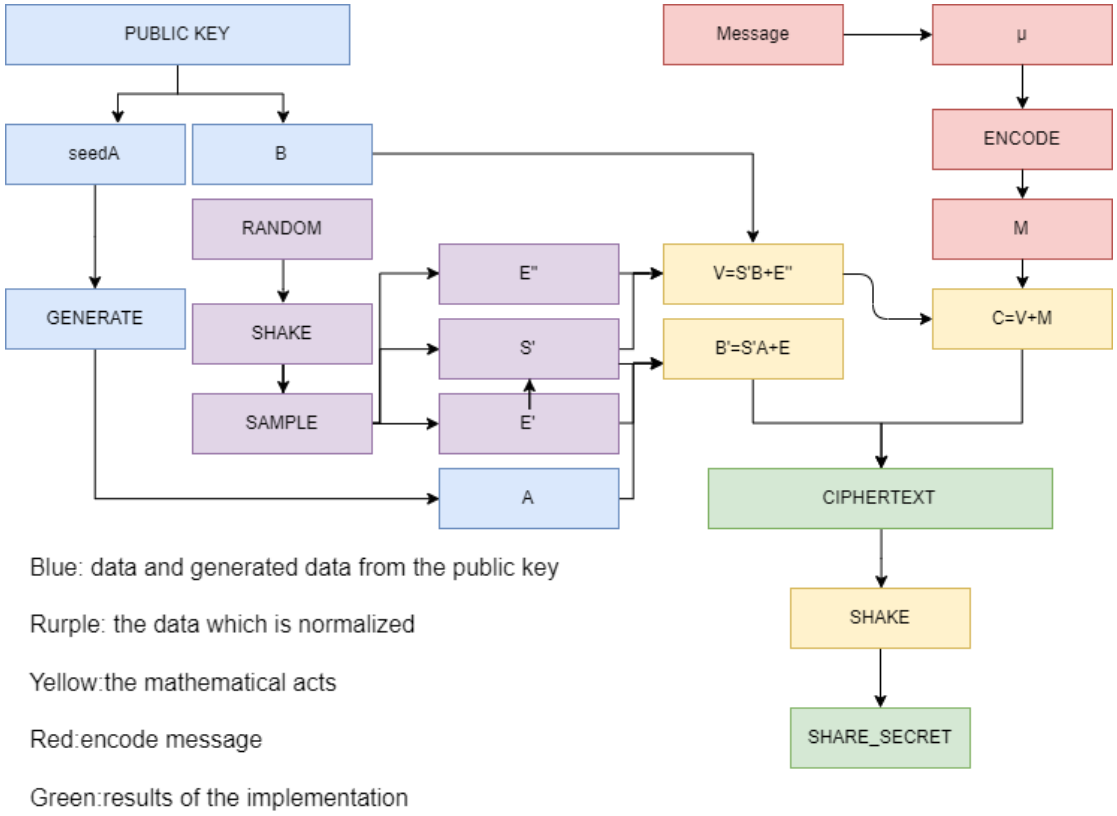


Figure 5: Share_secret_e and Cipher text creation step by step with the public key.

4.3 Decaps or Decryption

Decaps() or Decryption is required to ascertain the validity of the cipher text with the evaluation of the first share_secret. Decaps has the secret key, cipher text and share_secret as parameters and is named Decaps(secretkey,ciphertext,shresecret2). In the last stage, after the implementation of Decaps, it is required to check that share_secret1 is equal to the sharesecret2 and upon proving they are equal, then there is not an attack [2].

Figure 6, Figure 7 presents the stages in which Decaps() is implemented. Implementation begins with the unpacking of the two parts of the cipher text, c1 and c2 as B'=Frodo.Unpack(c1) and C=Frodo.Unpack(c2). Moreover, B' and C with a part of sk, which is called S, are computed, and they generate the array M, M=C-B'S, where M is processed with the Frodo.Decode(M) function and produces m. Pk includes the pk=seedA||b from the sky. After that it is important to generate two pseudorandom arrays with the function Frodo.SHAKE to create the seedse and a random bit of string [2].

This random bit of string is separated in three parts and each part is processed with the function Frodo.SampleMatrix which normalizes the arrays sampling the error matrix

and create the three arrays E', S' and E''. Furthermore, the system generates a new array which is called A. The implementation for this is to give the first 16 bit of the public key to the function of Frodo.Gen(seedA). After, the step is to compute the arrays A,S' and E' with the combination B''=AS'+E' . The following step is to unpack b, which is a part of the public key, and create the B, B=Frodo.Unpack(b). B, S' and E'' are computed and produced the V=S'B+E'' and V is used to produce array C', C'=V+Frodo.Encode(m). The last step is to check if B'or C is equal to B''or C'. If the arrays are equal the share_secret is created with the right variables, otherwise the share secret created comes with a variable error [2].

Algorithm 14 FrodoKEM.Decaps.

Input: Ciphertext $\mathbf{c}_1\|\mathbf{c}_2 \in \{0,1\}^{(\overline{m}\cdot n+\overline{m}\cdot\overline{n})D}$, secret key $sk' = (\mathbf{s}\|\mathsf{seed}_{\mathbf{A}}\|\mathbf{b},\mathbf{S}^{\mathrm{T}},\mathbf{pkh}) \in \{0,1\}^{\mathsf{len_s}+\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}+D\cdot n\cdot\overline{n}} \times \mathbb{Z}_q^{\overline{n}\times n} \times \{0,1\}^{\mathsf{len}_{\mathbf{pkh}}}$.

Output: Shared secret $\mathbf{ss} \in \{0,1\}^{\mathsf{len}_{\mathbf{ss}}}$.

1: $\mathbf{B}' \leftarrow \mathsf{Frodo.Unpack}(\mathbf{c}_1, \overline{m}, n)$
2: $\mathbf{C} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{c}_2, \overline{m}, \overline{n})$
3: Compute $\mathbf{M} \leftarrow \mathbf{C} - \mathbf{B}'\mathbf{S}$
4: Compute $\mu' \leftarrow \mathsf{Frodo.Decode}(\mathbf{M})$
5: Parse $pk \leftarrow \mathsf{seed}_{\mathbf{A}}\|\mathbf{b}$
6: Generate pseudorandom values $\mathsf{seed}_{\mathbf{SE}}{}'\|\mathbf{k}' \leftarrow \mathrm{SHAKE}(\mathbf{pkh}\|\mu', \mathsf{len}_{\mathsf{seed}_{\mathbf{SE}}} + \mathsf{len}_{\mathbf{k}})$
7: Generate pseudorandom bit string $(\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{m}\overline{n}-1)}) \leftarrow \mathrm{SHAKE}(\mathsf{0x96}\|\mathsf{seed}_{\mathbf{SE}}{}', (2\overline{m}n + \overline{m}\overline{n}) \cdot \mathsf{len}_{\chi})$
8: Sample error matrix $\mathbf{S}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(0)}, \mathbf{r}^{(1)}, \ldots, \mathbf{r}^{(\overline{m}n-1)}), \overline{m}, n, T_{\chi})$
9: Sample error matrix $\mathbf{E}' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(\overline{m}n)}, \mathbf{r}^{(\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n-1)}), \overline{m}, n, T_{\chi})$
10: Generate $\mathbf{A} \leftarrow \mathsf{Frodo.Gen}(\mathsf{seed}_{\mathbf{A}})$
11: Compute $\mathbf{B}'' \leftarrow \mathbf{S}'\mathbf{A} + \mathbf{E}'$
12: Sample error matrix $\mathbf{E}'' \leftarrow \mathsf{Frodo.SampleMatrix}((\mathbf{r}^{(2\overline{m}n)}, \mathbf{r}^{(2\overline{m}n+1)}, \ldots, \mathbf{r}^{(2\overline{m}n+\overline{m}\overline{n}-1)}), \overline{m}, \overline{n}, T_{\chi})$
13: Compute $\mathbf{B} \leftarrow \mathsf{Frodo.Unpack}(\mathbf{b}, n, \overline{n})$
14: Compute $\mathbf{V} \leftarrow \mathbf{S}'\mathbf{B} + \mathbf{E}''$
15: Compute $\mathbf{C}' \leftarrow \mathbf{V} + \mathsf{Frodo.Encode}(\mu')$
16: **if** $\mathbf{B}'\|\mathbf{C} = \mathbf{B}''\|\mathbf{C}'$ **then**
17: **return** shared secret $\mathbf{ss} \leftarrow \mathrm{SHAKE}(\mathbf{c}_1\|\mathbf{c}_2\|\mathbf{k}', \mathsf{len}_{\mathbf{ss}})$
18: **else**
19: **return** shared secret $\mathbf{ss} \leftarrow \mathrm{SHAKE}(\mathbf{c}_1\|\mathbf{c}_2\|\mathbf{s}, \mathsf{len}_{\mathbf{ss}})$

Figure 6: Algorithm FrodoKEM.Decaps. (as appended to Alkim et al., 2021, p. 22)

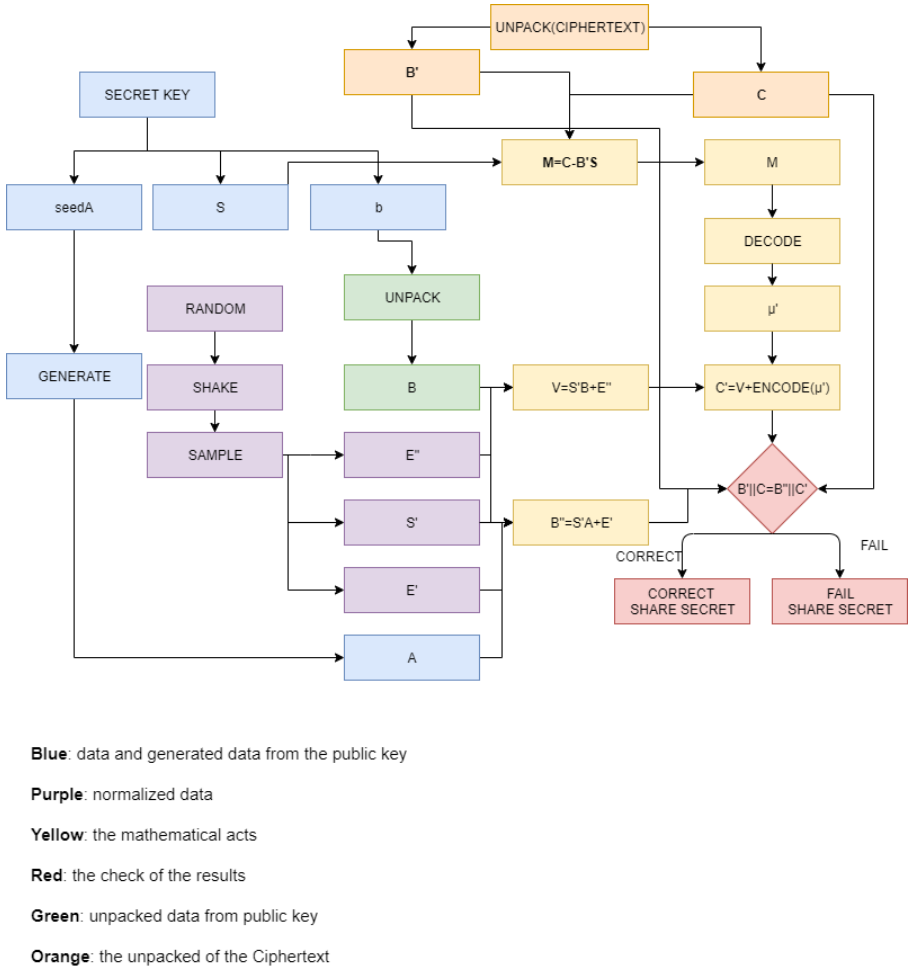


Figure 7: Implementation of Decap or decryption with the secret_key and create the share_secret_d

4.4 Generating Matrix A

To generate Matrix A there are two options, the first one uses AES128 and the other uses SHAKE128 [2].The Figure 8,Figure 9 depict the process which generates A<-Frodo.Gen(seedA). In my thesis, between the two different Arm Cortex M type processors SHAKE128 proved quicker than AES128.

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$.
Output: Pseudorandom matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.

1: **for** $(i = 0; i < n; i \leftarrow i + 1)$ **do**
2: $\quad \mathbf{b} \leftarrow \langle i \rangle \| \mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{16+\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$ where $\langle i \rangle \in \{0,1\}^{16}$
3: $\quad \langle c_{i,0} \rangle \| \langle c_{i,1} \rangle \| \cdots \| \langle c_{i,n-1} \rangle \leftarrow \mathrm{SHAKE128}(\mathbf{b}, 16n)$ where each $\langle c_{i,j} \rangle \in \{0,1\}^{16}$
4: $\quad$ **for** $(j = 0; j < n; j \leftarrow j + 1)$ **do**
5: $\quad\quad \mathbf{A}_{i,j} \leftarrow c_{i,j} \bmod q$
6: **return** $\mathbf{A}$

Figure 8: Algorithm FrodoKEM.Gen using SHAKE128 (as appended to Alkim et al., 2021, p. 17)

Input: Seed $\mathsf{seed}_{\mathbf{A}} \in \{0,1\}^{\mathsf{len}_{\mathsf{seed}_{\mathbf{A}}}}$.
Output: Matrix $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$.

1: **for** $(i = 0; i < n; i \leftarrow i + 1)$ **do**
2: **for** $(j = 0; j < n; j \leftarrow j + 8)$ **do**
3: $\mathbf{b} \leftarrow \langle i \rangle \| \langle j \rangle \| 0 \cdots 0 \in \{0,1\}^{128}$ where $\langle i \rangle, \langle j \rangle \in \{0,1\}^{16}$
4: $\langle c_{i,j} \rangle \| \langle c_{i,j+1} \rangle \| \cdots \| \langle c_{i,j+7} \rangle \leftarrow \mathrm{AES128}_{\mathsf{seed}_{\mathbf{A}}}(\mathbf{b})$ where each $\langle c_{i,k} \rangle \in \{0,1\}^{16}$
5: **for** $(k = 0; k < 8; k \leftarrow k + 1)$ **do**
6: $\mathbf{A}_{i,j+k} \leftarrow c_{i,j+k} \bmod q$
7: **return** $\mathbf{A}$

Figure 9: Algorithm FrodoKEM.Gen using AES128(as appended to Alkim et al., 2021, p. 17)

5. STM board design

This chapter describes the hardware features that each microcontroller uses to run the algorithm of FrodoKEM. These hardware features are different depending on the microcontroller and helped execute the algorithm. Some basics characterized are the RNG, UART, FLASH and etc.

5.1 STM32 Programming environment

STM develops several microcontrollers for different domains. The first development environment used in this thesis is STM32CubeIDE. The tools included assist in developing an application code for such microcontrollers, compile, debug, and run the application [21]. Additionally, STM32CubeProgrammer was used to help with the monitoring of the memory when the program was loaded on the microcontroller.

5.2 STM32L552ZE

STM32L552 contains many hardware features which can help the program to be flexible and quick. Among the hardware features which were used,there were: the Art-Accelerator which is a type of cache memory; RNG based on NIST which generates random numbers; LPUART which supports asynchronous serial communication with minimum power consumption; along with DEBUG, PWR, GPIO and SYS; and the timer TIM-2 which has a 32-bit auto-reload up/down-counter [21]. Apart from including TrustZone and offering an ultra-low power Arm Cortex-M33 32-bit MCU which has a 110MHz frequency, there is up to 512KB space in Flash memory and there is up to 256KB SRAM space, as well.

ART-ACCELERATOR: is the basic accelerator which helps the programs to change the duration of the process. To be more precise, Art accelerator or instruction cache (ICACHE) is introduced on the C-AHB code bus of Cortex®-M33 processor to improve performance when fetching instruction (or data) from both internal and external memories, as those of SRAM and Flash. Apart from that, cache supports an 8-Kbyte instruction cache with frequency up to 110 MHz and is used in three different ways. The first option is to enable cache in one-way mode. The second one is to enable it in a two-way mode, and the last one is to disable it [21]. This process results in having better ascription in running time depending on the usage of cache.

It is important to mention the difference between one-way or two-way cache. In the two-way set, the associative mode features 256 lines of 16 bytes. As a result, the 4 LSbs of the address reflect a cache line offset, and the 8-bit index picks one entry from 256 lines in the tag and data memory. In the direct-mapped (one-way), there are 512 lines of 16 bytes and the index consists of 9-bit. The basic differences between the one-way and two-way modes involve the low power consumption in the direct-mapped and the absence of a replacement algorithm in the case of a cache line eviction in the direct-mapped setting.

DEBUGGER: The debug mode has been set to 'Trace Asynchronous SW'. More specifically, when that debugging mode is enabled, debug operates using some pins, namely PA13, PA14, and PB3 from GPIO. In addition, this mode supports tools, like SWV which, among others, helps dealing with monitoring time, memory and variables [21]. For this thesis, an application Data Trace is used, which helps to count the time and authenticate the time of the timer.

LPUART: refers to the connection of the console. This means that under certain circumstances, the user is able to print arrays and messages. LPUART is a single low-power UART that allows for asynchronous serial communication with low power consumption. Apart from that, it provides support for communication with a half-duplex single wire which uses only Tx for transmission and reception and there is a full-duplex which connects the Tx with the Rx, and other operations like CTS, RTS. LPUART's clock is independent of the CPU's and can restart the system if it is turned off. Communication has a maximum baud rate of 9600 baud [21]. The device consumes less power because even while in standby mode, LPUART may wait for a frame while consuming minimal power. DMA controller can transmit data to LPUART, and the data length can range from 7 or 8 to 9 bits.

Activating LPUART in STM32L552 and connecting it with the console requires some steps. First, LPUART must be active to be put into asynchronous mode in order to get the baud rate to 115200Bit/s and word length to 8 bits [21].Also, to transmit the data to the console and print them, the basic function to be added into the program is HAL_UART_Transmit; a simple way to print is a char array due to the fact that there is no need to change the data form. To print integers, it is necessary to reshape the int to char with the help of sprintf.

```
void print_int(uint8_t k)
{
    char pin[20];
    sprintf(pin,"%02x",k);
//  HAL_UART_Transmit(&huart2,k,1, 2000);
    HAL_UART_Transmit(&hlpuart1,(uint8_t*)pin,strlen(pin), 2000);
}
```

Figure 10: The code of LPUART print

TIMER TIM-2: is a timer of general purpose which has the highest counter resolution. More specifically, there is a 32-bit auto-reload up/down-counter and a 32-bit prescaler. In debug mode, when the program is frozen the timer stops counting [21]. In this program, the clock source comes from the internal clock, and the timer prescaler is based on a frequency of 1,100 MHz.

```
cnt=0;
htim2.Instance->CNT=0;
HAL_TIM_Base_Start(&htim2);
rc =PQCLEAN_FRODOKEM640SHAKE_OPT_crypto_kem_keypair(public_key, secret_key);
HAL_TIM_Base_Stop(&htim2);
h=htim2.Instance->CNT;
```

Figure 11: Code of how to count the time with Timer TIM-2

More specifically, timer TIM2 is used to count the time in the functions of the programs. In the beginning, the timer code is defined through the command (htim2.Instance->CNT=0;) htim2 equal to zero. The next step is to get the timer to start with the function of HAL, HAL_TIM_Base_Start(&htim2) by adding the command HAL_TIM_Base_Stop(&htim2) when there is the need to stop counting. The last step is the command h=htim2.Instance->CNT to retrieve the counted clock cycles. The result h is in the form of clock cycles and to convert it to seconds the division h/1,000,000 is implemented [21].

RNG-RANDOM GENERATOR: is a true random number generator that provides full entropy outputs to the application as 32-bit samples. It is composed of a live entropy source (analog) and an internal conditioning component. Furthermore, RNGs used for cryptographic applications typically produce sequences made of random 0's and 1's bit which are non-deterministic, which means that the random generator produces randomness that depends on some unpredictable physical source.[21].

RNG implementation is based on an analog circuit and is produced in stm32 MCUs. This circuit generates a continuous analog noise that is used to generate a 32-bit random number during the RNG process. The analog circuit consists of many ring oscillators with XORed outputs. RNG processing is clocked at a consistent frequency using a dedicated clock, which can be decreased using the divider inside the RNG peripheral for a subset of microcontrollers[21].

```
int randombytes(uint8_t *buf, size_t n) {
#if defined(STM32L552xx)
    uint32_t p[n];
    int i,j=0;
    for(i=1;i<=n/4;i++)
    {
     HAL_RNG_GenerateRandomNumber(&hrng,(p+i));
       for(j=j;j<i*4;j++)
       {
          buf[j] = (uint8_t)p[i];
          j=j+1;
          buf[j] = (uint8_t)(p[i]>>8);
          j=j+1;
          buf[j] = (uint8_t)(p[i]>>16);
          j=j+1;
          buf[j] = (uint8_t)(p[i]>>24);
       }
    }
    return 1;
#else
#error "randombytes(...) is not supported on this platform"
    #endif
}
```

Figure 12: Create the random generated data with the hardware feature RNG

At first, for the needs of this thesis, a function called int randombytes(uint8_t *buf,size_t n);was created. The function has been recreated and with the help of the RNG it fills the matrix (uint8_t *buf). In Figure 12, there is a screen capture of the function using RNG. In a detailed description of randombytes, random bytes are generated by the basic function of HAL. For this command the name in NRG is HAL_RNG_GenerateRandomNumber. It inputs a uint32_t variable (array uint32_t p[n]), meaning that it generates 32 bit each time. For this reason, due to the data used in the main function in the form of arrays uint8_t (buff) inside the program, it is important to reshape data, for this occasion with the second *for* uint32_t (p) variable data is separated into four uint8_t (buff).

SRAM: provides 256KB which is split into three blocks with different addresses (ST life.augmented, 2020). The first and largest part of SRAM has 192KB mapped at address 0x20000000 called SRAM1. The smallest part, which is 64KB, is located at address 0x0A030000 with hardware parity check with the name of SRAM2. SRAM2 is also mapped at address 0X20030000, offering a contiguous address space along with SRAM1. This block is accessed through C-bus for maximum performance. In standby mode, the 64KB or the upper 4KB of SRAM2 can be retained. SRAM2 can be write-protected with 1 Kbyte granularity. Memory can be accessed at CPU clock speed in read/write mode with no wait states. Another option to achieve that all SRAMs are secure after a reset is to activate the TrustZone security. Besides this method of securing SRAMs, there is another choice, that of non-secure programming by block-based coding using the MPCBB (i.e, block-based memory protection controller) in GTZC controller. The granularity of the secure block-based RAM is a page of 256 bytes.

SRAM1 has all the static matrixes of the program and undertakes all the demanding work of running the application and keeping the data of the program safe without missing it[21]. For this reason, in FrodoKEM640 either with AES or SHAKE128, there are 3 static arrays: the public_key with 9616B, the secret_key 19888B, and the last one is the ciphertext with 9720B. In the main functions of FrodoKEM, there is the

need to handle an extra memory during the Decryption function requiring 78 KB. In the stage of encryption, the space needed is up to 58 KB. The last function, Keypair, the space needed is up to 36KB. That means that it is necessary to adjust the heap and stack memory. Also, what is indispensable to achieve greater flexibility in embedded systems is further understanding how memory works and apply certain specific actions as far as the code is concerned.

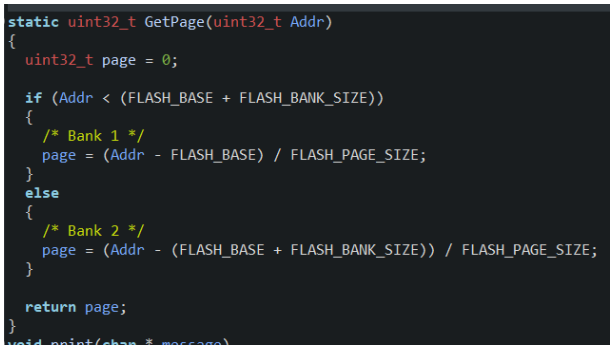
FLASH MEMORY: has 512 Kbytes which is available for storing programs and data [21]. Flash interface features are divided into two options: the first is Single or dual bank operating modes and the second is Read-while-write (RWW) in dual bank mode. Both of them allow for a read operation in one bank while performing an erase or program operation in the other. Dual bank boot is also available. Each bank has 128 pages of 2 or 4 kilobytes each, depending on the read access width. Flash memory also embeds one-time programmable 512 bytes OTP for user data.

Apart from that, embedded flash memory supports flexible protection techniques which can be configured owing to option bytes [21]. One of them is readout protection which can protect the whole memory and has four different levels of protection. The second one is write protection which protects a specific area. This specific area is protected against erasing and programming with two different modes available: single bank mode and dual bank mode. The most significant feature is that it is a non-volatile memory and embeds the error correction code. Another feature of Flash memory is TrustZone security which, after resetting, secures Flash memory.

Most of the time, a microcontroller has to handle many processes and an algorithm like FrodoKEM640AES which needs much space in memory. For instance, as for FrodoKEM640AES, the function of Decryption needs 142KB to run but SRAM1 has only 192KB free space, meaning that it is important to use Flash and store some important details there due to the reduction of space in SRAM1 memory. Applications ought to use FrodoKEM640 as a tool for security and therefore avoid causing space issues occasioned by the creation of a function transmitting the main data to Flash.

RamtoFlash is the name of a function that includes two functions: one that erases the pages in Flash and another that copies data from an array to Flash pages [21].Cache must be disabled before erasing the pages, and then the primary two functions must be started before cache can be activated again. To delete the pages, three variables must be provided: FirstPage which contains the first page; NbOfPages which contains the number of pages; and BankNumber which contains the first page. The variables must then be filled into the EraseInitStruct, and Flash memory must be erased. The cache can then be enabled after programming the user Flash area word by word.

Figure 13: The function RamtoFlash send the data from the SRAM to Flash



```
static uint32_t GetPage(uint32_t Addr)
{
  uint32_t page = 0;

  if (Addr < (FLASH_BASE + FLASH_BANK_SIZE))
  {
    /* Bank 1 */
    page = (Addr - FLASH_BASE) / FLASH_PAGE_SIZE;
  }
  else
  {
    /* Bank 2 */
    page = (Addr - (FLASH_BASE + FLASH_BANK_SIZE)) / FLASH_PAGE_SIZE;
  }

  return page;
}
```

Figure 14: GetPage is an function which used by RamtoFlash

5.3 STM32WL55JC1

STM32WL55JC1 is a microcontroller with many hardware features which can help the program to be flexible and quick, and create the .ioc file [16]. The microcontroller enables the ART-Accelerator; RNG based on NIST generates random numbers; USART2 support provides asynchronous communication along with AES-accelerator and some features, like PWR, GPIO and SYS; and TIM-2 which has a 32-bit auto-reload up/down counter.

The other features provide a dual core 32-bit Arm Cortex-M4 and Arm Cortex-M0 which has a frequency up to 48MHz; Flash memory has space up to 256KB and SRAM has space up to 64KB. In this thesis, Arm-Cortex-M4 was chosen. Some of the hardware features are the same as in the microcontroller STM32L552, namely two of them are RNG and TIM-2 and are referred to in TIMER-TIM2 and RNG-Random generator system (chapter 5.2 STM32L552*RNG*).

ART-ACCELERATOR: Processor Arm Cortex-M4 contains the memory accelerator under the name of ART which is designed for this specific processor [16]. ART increases the frequencies of Flash memory and due to the accelerator balance is created between the frequency of Flash memory and the frequency of the processor. Arm Cortex-M4 does not have to wait in high frequencies for the flash memory.

ART Accelerator uses an instruction prefetch queue and branch cache to boost program execution speed from the 64-bit Flash memory, allowing the processor to achieve around 60 DMIPS performance at 48 MHz. According to the CoreMark benchmark, the ART Accelerator's performance is equivalent to the execution of a 0 wait state application from Flash memory at a CPU frequency of up to 48 MHz. Another detail is that ART-Accelerator is enabled in the microcontroller from the beginning.

EMBEDDED FLASH MEMORY: interface controls the access to Flash memory from CPU1 AHB ICode/DCode and CPU2 AHB Sbus. It implements read and write protection, as well as access, erase, and program Flash memory operations [16].The following are the primary characteristics of Flash memory:

• Organizing your memory: 1 bank – main memory up to 256 KB – page size 2 KB

• Data read with a 72-bit width (64 bit plus 8 ECC bit)

• Data write with a 72-bit width (64 bit plus 8 ECC bit)

• Erasing a page and erasing a group of pages

Flexible safeguards, which can be customized via option bytes, are an added bonus. The readout protection (RDP) is used to safeguard the entire memory. There are two levels to choose from. The initial level is level 0, which has no readout protection. The other is level 1, which protects against memory readout. If debug features are connected, boot in SRAM or bootloader is selected, the Flash memory cannot be read or written. The final degree of protection is level 2, which protects against chip readout [16].

EMBEDDED SRAM: The devices feature up to 64 Kbytes of embedded SRAM, split in two blocks [16]:

• SRAM1: up to 32 Kbytes mapped at address 0x2000 0000

• SRAM2: up to 32 Kbytes located at address 0x2000 8000 (contiguous to SRAM1), also mirrored at 0x1000 0000, with hardware parity check - this SRAM can be retained in standby mode.

Access to SRAMs can take place in read/write with 0 wait states for every CPU1/2 clock speed.

USART-2: is a universal synchronous receiver-transmitter which provides asynchronous communication [16]. Except that, it supports IrDA SIR ENDEC and a multiprocessor communication mode. Another feature is that it includes single-wire half-duplex communication mode. Moreover, USART-2 has LIN Master/Slave capability and provides hardware management for the CTS and RTS signals, and RS485 driver enable.

USART-2 frequency, while being able to communicate at an up to 4 Mbit/s speed, also offers the Smart Card mode and SPI-like communication capability. USART-2 supports synchronous operation thanks to SPI and allows for the capability to be used as an SPI master. The clock of the CPU is independent from the clock of USART-2, allowing the USART to perform the wakeup for MCU from stop mode, using baud rates up to 200 kbaud. USART-2 includes wake up events from stop mode and can be programmed with three options. The first one is the start bit detection, the second any received data frame, and the last a specific programmed data frame. The DMA controller is able to run the USART interface.

USART-2 has the same code as LPUART. However, the only difference is that instead of hlpuart1, huart2 is used. LPUART is referred to in LPUART, chapter 4.2.2 STM32L552.

AES-ACCELERATOR: In this thesis, the AES accelerator is used to reduce the time of the program. AES encrypts and decrypts data using an algorithm and implementation that are completely consistent with FIPS (2001).For key sizes of 128 or 256 bits, a variety of chaining modes (ECB, CBC, CTR, GCM, GMAC, CCM) are available. In this thesis, ECB mode with 128B key size is used.

```
/* USER CODE END AES_Init 1 */
hcryp.Instance = AES;
hcryp.Init.DataType = CRYP_DATATYPE_8B;
hcryp.Init.KeySize = CRYP_KEYSIZE_128B;
hcryp.Init.pKey = (uint32_t *)pKeyAES;
hcryp.Init.Algorithm = CRYP_AES_ECB;
hcryp.Init.DataWidthUnit = CRYP_DATAWIDTHUNIT_BYTE;
hcryp.Init.HeaderWidthUnit = CRYP_HEADERWIDTHUNIT_BYTE;
hcryp.Init.KeyIVConfigSkip = CRYP_KEYIVCONFIG_ALWAYS;
if (HAL_CRYP_Init(&hcryp) != HAL_OK)
{
  Error_Handler();
}
/* USER CODE BEGIN AES_Init 2 */
```

Figure 15: How to init the AES accelerator

```
trix_aes.c | C util.c | C common.h | C kem.c | STM32WL55JCIX_FLASH.ld | C main.c
/* USER CODE END 0 */

CRYP_HandleTypeDef hcryp;
__ALIGN_BEGIN static const uint32_t pKeyAES[4] __ALIGN_END =
{0x2b7e1516,0x28aed2a6,0xabf71588,0x09cf4f3c};
```

Figure 16: How to write AES Key with 128bits

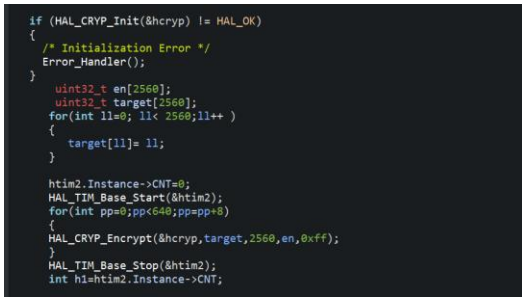


Figure 17: How to implement theAES code

Figure 15, Figure 16, Figure 17 show how the algorithm works. At first, the user has to program how they prefer the algorithm of AES to run. In this case, the strategy is to use 8 bits of the data each time and a 128bit key. Except that, the AES type of ECB is used. The second step is to write the key in the pKeyAES[4] area and, last, to run the function HAL_CRYP_Encrypt.

6. Algorithm implementation

This chapter refers to the implementation of the algorithm in two different microcontrollers the STM32L552ZE and the STM32WL55JC1. In the first microcontroller, the algorithm is tested for the performance of time depending on the type of cache used. While in the second microcontroller due to the small memory SRAM different techniques are used to reduce the space used by the algorithm.

6.1 STM32L552ZE Introduction

Ten separate programs were developed in STM32L552, each considering different hardware features of the board. The first five programs include FrodoKEM640-AES and the second five include FrodoKEM640-SHAKE as their main code and are implemented according to the usage of flash and usage of cache. As a result, each program requires different run times in implementing each function, allowing for tracking and analyzing their differences:

1. FRODOKEM640-AES with 1-way cache
2. FRODOKEM640-AES with 2-way cache
3. FRODOKEM640-AES without cache
4. FRODOKEM640-AES with Flash and 1-way cache
5. FRODOKEM640-AES with Flash and 2-way cache
6. FRODOKEM640-SHAKE with 1-way cache
7. FRODOKEM640-SHAKE with 2-way cache
8. FRODOKEM640-SHAKE without cache
9. FRODOKEM640-SHAKE with Flash and 1-way cache
10. FRODOKEM640-SHAKE with Flash and 2-way cache

Every program has basic hardware features. These features are RNG, timer TIM2, DEBUGGER and LPUART. Differences in the programs include the case where cache is enabled or when the program uses flash.

6.2 Basic implementation for FrodoKEM640-AES and FrodoKEM640-SHAKE

FrodoKEM640-AES and FrodoKEM640-SHAKE at first begin with the creation of the two keys, the private key, and the secret key. To create the two basic keys, 48B needs to be generated which is produced by the function of the randombytes (chapter 5.2 STM32L552ZE*RNG*) and it is also produced at 16B encryption which produces the share_secret.

LPUART is useful to print the details and to understand how the system operates each time. It is helpful to make comments inside the code and print them in the terminal. Another option is to use LPUART and print the keys and save them in an .h file in Flash.

Flash memory is the memory where the code is when the program starts up. Apart from that, the global symbols are in flash too. Furthermore, in .h files with the keys and ciphertext, the data is initialized with the word const. The const data is stored in Flash and this can significantly reduce the space in SRAM1. Another option to use flash is to enter data in flash and read it with a pointer and this also helps to reduce the space in the memory of SRAM1.

After the loading of the program in flash, the program runs in SRAM1. First, it is useful to have a static matrix in the program. It helps to reduce the arrays and also the space of memory. Also, it is SRAM1 which when not combined with other types of memory makes the program quicker.

PUBLIC KEY	SECRET KEY	CIPHERTEXT	SHARE SECRET ENCRYPTION	SHARE SECRET DECRYPTION
9616B	19888B	9720B	16B	16B

Table 1: Basic variables for running (used each time)

FrodoKEM640-AES with one-way cache: FrodoKEM640-AES used many of the hardware features. RNG, TIM2, and cache are enabled in one-way or direct mapped; DEBUG mode and the LPUART are also enabled. There is no need for the program to be separated into parts and run because SRAM1 can handle each process without the help of Flash.

FrodoKEM640-AES with two-way cache: FrodoKEM640-AES used many of the hardware features. RNG, TIM2, and cache are enabled in two-way; DEBUG mode

and LPUART are also enabled. There is no need for the program to be separated into parts and run because SRAM1 can handle each process without the help of Flash.

FrodoKEM640-AES without cache: FrodoKEM640-AES used many of the hardware features, which are RNG, TIM2, DEBUG mode, and LPUART. There is no need for the program to be separated into parts and run because SRAM1 can handle each process without the help of Flash.

	KEYPAIR	ENCRYPTION	DECRYPTION
MEMORY	41160B	82032B	102680B

Table 2: FrodoKEM640-AES with one-way cache, two-way cache, without cache

FrodoKEM640-AES with one-way cache: FrodoKEM640-AES used many of the hardware features. RNG, TIM2, and cache are enabled one-way; DEBUG mode and LPUART are also enabled. To reduce the space of memory and handle and other processes, it is necessary to use Flash and store the keys and the ciphertext there with the program decreasing the memory space by 39KB.

FrodoKEM640-AES with two-way cache: FrodoKEM640-AES with two-way cache: FrodoKEM640-AES used many of the hardware features. RNG, TIM2, and cache are enabled two-way; DEBUG mode and LPUART are also enabled. To reduce the space of memory and handle and other processes, it is necessary to use Flash and store the keys and the ciphertext there with the program decreasing the memory space by 39KB.

	KEYPAIR	ENCRYPTION	DECRYPTION
MEMORY	30936B	51832B	50064B

Table 3: FrodoKEM640-AES with one-way cache, two-way cache, Flash

FrodoKEM640-SHAKE with one-way cache: FrodoKEM640-SHAKE used many of the hardware features. RNG, TIM2, and cache are enabled in one-way; DEBUG mode and LPUART are also enabled. There is no need for the program to be separated into parts and run because SRAM1 can handle each process without the help of Flash.

FrodoKEM640-SHAKE with two-way cache: FrodoKEM640-SHAKE used many of the hardware features: RNG, TIM2, and cache are enabled in two-way; DEBUG mode and LPUART are also enabled. There is no need for the program to be separated into parts and run because SRAM1 can handle each process without the help of Flash.

FrodoKEM640-SHAKE without cache: FrodoKEM640-SHAKE used many of the hardware features. RNG, TIM2; DEBUG mode and LPUART are also enabled. There

is no need for the program to be separated into parts and run because SRAM1 can handle each process without the help of Flash.

	KEYPAIR	ENCRYPTION	DECRYPTION
MEMORY	36064B	57728B	78376B

Table 4: FrodoKEM640-SHAKE with one-way cache, two-way cache, without cache

FrodoKEM640-SHAKE with one-way cache and Flash: FrodoKEM640-AES used many of the hardware features. RNG, TIM2, and cache are enabled in one-way or direct mapped; DEBUG mode and the LPUART are also enabled. To reduce the space of memory and handle other processes, it is necessary to use FLASH and store the keys and ciphertext there with the program decreasing the memory space by 39KB.

FrodoKEM640-SHAKE with two-way cache and Flash: FrodoKEM640-AES used many of the hardware features. RNG, TIM2, and cache are enabled in one-way or direct mapped; DEBUG mode and the LPUART are also enabled. To reduce the space of memory and handle other processes, it is necessary to use FLASH and store the keys and ciphertext there with the program decreasing the memory space by 39KB (see 4.3.2 STM32WL55JC1)

	KEYPAIR	ENCRYPTION	DECRYPTION
MEMORY	25840B	27256B	31448BB

Table 5: FrodoKEM640-SHAKE with one-way cache, two-way cache with Flash

6.3 STM32WL55JC1 introduction

Three separate programs were developed in STM32WL55JC1, each considering a different algorithm. The first program includes the FrodoKEM640-AES and the second includes FrodoKEM640-SHAKE. The third program is based on AES-ECB with 128B key size. To reduce bit size, the first two programs have to implement a separate function of the FrodoKEM640-AES and FrodoKEM640-SHAKE and generate different times. AES ECB encrypts 10KB and counts the time which is needed.

Algorithms used in the three programs:

1. FrodoKEM640-AES
2. FrodoKEM640-SHAKE
3. AES ECB 128B

Every program has basic hardware features. These features are: RNG, timer TIM2, and USART-2. The challenge in this program is to reduce the space needed for every function.

6.4 Basic implementation of FrodoKEM640-AES and FrodoKEM640-SHAKEon STM32WL55JC1

The FrodoKEM640-AES and FrodoKEM640-SHAKE at first begin with the creation of the two keys, the private key, and the secret key. To create the two basic keys 48B need to be generated which are produced by the function of randombytes (chapter 5.2 STM32L552ZERNG) and are also produced at 16B encryption which produces the share_secret.

The programs are separated in parts every time #if is used to run only the function of Encryption or Decryption or Keypair. The results of encryption and Keypair are placed at Flash with the help of an .h file.

USART-2 is useful to print the details and to understand how the system operates each time. It is helpful to make comments inside the code and print them in the terminal. Another option is to use the USART2 and print is to print to the keys and save them in an .h file in Flash.

Flash memory is the memory where the code is when the program starts up. Apart from that, the global symbols are in flash, too. Furthermore, in .h files with the keys and ciphertext, the data is initialized with the word const. The const data is stored in Flash and this can significantly reduce the space in RAM. Another option is to use the flash by entering data in flash and to read it with a pointer and this also helps to reduce the space in the memory of SRAM.

After the loading of the program in flash, the program runs in SRAM1. First, it is useful to have a static matrix in the program. It helps to reduce the arrays and also the space of memory. Also, it is SRAM1 which when not combined with other types of memory makes the program quicker.

AES ECB is useful because the user can encrypt data with the accelerator, resulting in the least amount of time there can be, with the accelerator without the AES ECB software.

FrodoKEM640-AES used many of the hardware features: RNG, TIM2 and USART-2. The program needs to be separated into parts and run because SRAM1 cannot handle each process without the help of the Flash.

	KEYPAIR	ENCRYPTION	DECRYPTION
MEMORY	30936B	51832B	50064B

Table 6: FrodoKEM640-AES

The FrodoKEM640-SHAKE used many of the hardware features. The RNG, TIM2, and USART-2. The program needs to be separated into parts and run because SRAM1 cannot handle each process without the help of the Flash.

	KEYPAIR	ENCRYPTION	DECRYPTION
MEMORY	25840B	27256B	31448B

Table 7: FrodoKEM640-SHAKE

AES ECB-128B: used AES ACCELERATOR and Flash as main tools. The AES ECB program is used to make some counts of the time while AES encryption runs.

6.5 Technique and spacereduction

The techniques to reduce the space were three:

1. Preprocess the keys
2. Include data in Flash in specified space with the function RamtoFlash
3. Static arrays

Preprocess the keys and static arrays: to preprocess the data it is needed to create an .h file with the keys. To generate the keys, it is important to use the feature ofLPUART. The first step is to run the FrodoKEM640-AES or FrodoKEM640-SHAKE keypair function and then print with the function of printing the private and public key. The next step is to create an .h file and insert the include #include “stdint.h” and initialize the arrays as uint8_t and const. An explanation for the word const is that defines the data as constant and the constant data is stored in Flash due to the linker script.After, the next step is to paste the public and secret key there. For the ciphertext it is necessary to run the encryption function and to implement the same steps. This technique creates three static arrays in this thesis and there are the following arrays. These three arrays are: public key 9616B, secret key 19888B, and ciphertext 9720B.

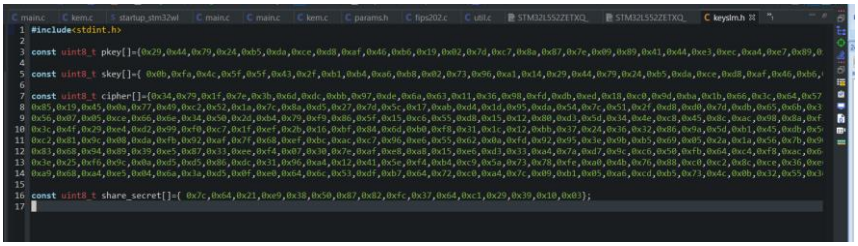


Figure 18: Preprocess keys and Ciphertext in a .h file

Shared arrays&static arrays: a shared table is a table that you do not initialize with values when you first declare it and it has a lifespan until the end of the program run. It has a local scope. The table is not initialized, so its elements are initialized to the default value which is zero for the arithmetic formulas uint16_t. In more detail, this table is used in place of a different variable and each time it ends the use in which it is used, the data is transferred either to Flash or the corresponding operations are performed that aim at the implementation of the algorithm.

To be more precise, the function of Encryption, Decryption and Keypair need a huge amount space in the memory to run. The function of Keypair uses 30864B to run, the function of Encryption requires 51248B and the function of Decryption demand space up to 71896B. Therefore, the space requirements of the algorithm are sufficient. For this reason, there is a shared array which consumes only 20KB of SRAM and with the transfer of data in Flash it helps reducing the space in SRAM.

RamtoFlash: the necessary function Flash to Ram is used to embed data in Flash (see Embedded Flash Memory for analysis of Flash and how it works in 5.2 STM32L552ZE). Herein, the focus of the analysis is placed on how Flash is used for the functions: keypair, encryption, and Decryption.

6.6 RamtoFlash - Static arrays - .h files - Keypair - AES640

CODE	CODE CHANGES	THEORY
uint_8t pk[9616]	uint8_t pk[9616]	Public key
uint8_t *pk_seedA=&pk[0]	uint8_t *pk_seedA=&pk[0]	seedA
uint8_t *pk_b=&pk[16]	uint8_t *pk_b=&pk[16]	b
uint8_t sk[19888]	uint8_t sk[19888]	Secret key
uint8_t *sk_s=&sk[0]	uint8_t *sk_s=&sk[0]	
uint8_t *sk_pk=&sk[16]	uint8_t *sk_pk=&sk[16]	
uint8_t *sk_S=&sk[9632]	uint8_t *sk_S=&sk[9632]	
uint8_t *sk_pkh=&sk[19872]	uint8_t *sk_pkh=&sk[19872]	pkh
uint16_t B[5120]	uint8_t *B=&sk[0]	B=A(B)S+E
uint16_t S[10240]	uint16_t S[10240]	S
uint16_t *E=&S[5120]	uint16_t *E=&S[5120]	E
uint8_trandomness[48]	uint8_trandomness[48]	

uint8_t *randomness_s=&randomness[0]	uint8_t *randomness_s=&randomness[0]	s
uint8_t *randomness_z=randomness[32]	uint8_t *randomness_z=randomness[32]	z
uint8_t SHAKE_input_seedSE[17];	uint8_t SHAKE_input_seedSE[17];	
uint8_t randomness_seedSE[17]	uint8_t randomness_seedSE[17]	seedSE

Table 8: Variables in code and Variables in theory Keypair()

The function Keypair has many variables and arrays: public key, secret key, two huge arrays (i.e., B and S), and three small arrays, randomness, SHAKE_input_seedSE, and randomness_seedSE. The public key and private key are required to bind 29504B space. Moreover, B and S need to occupy 30720B space, and the smaller variables need 81B. The two variables, B and S, have different roles in this function due to the fact that it is easy to delete the smaller one, which is B, and S is about 20K. The only variable transferred from Ram to Flash is S. Then, to read the variable from Flash, you only need to point a pointer of uint16_t to the memory location where the variable is located and to use this pointer instead of the variable S.

```
RamtoFlash(S,2 * PARAMS_N * PARAMS_NBAR,ADDR_FLASH_PAGE_40,ADDR_FLAS

uint32_t Address = ADDR_FLASH_PAGE_40;
uint16_t *Sdata;
Sdata=(__IO uint64_t*)Address;

uint16_t *E1= &Sdata[PARAMS_N * PARAMS_NBAR];               // conta

PQCLEAN_FRODOKEM640AES_OPT_mul_add_as_plus_e(S, Sdata, E1, pk);

// Encode the second part of the public key
    PQCLEAN_FRODOKEM640AES_OPT_pack(pk_b, CRYPTO_PUBLICKEYBYTES - BY

    for(int i=0;i<2 * PARAMS_N * PARAMS_NBAR;i++)
    {
        S[i]=Sdata[i];
```

Figure 19: Send the data of S in the Flash and create a pointer in Flash to read the data with a pointer

Figure 19 shows an example of how S can be read from Flash. The new variable uint16_t is the pointer Sdata, which takes the initial address of the array S[0], in the next line. The E1 pointer in the next line points to the variable at S[5.120]. Then S can be used instead of B and the user can write the results of the mul_add_as_plus_efunction there. The following step is to pack the data into the public key and load the data's array from Sdata to S. As a result, the amount of space available is reduced by 10240B.

In the end of the function, to keep the keys it is necessary to transfer the data in Flash. One option is to use RamtoFlash and another option is to create an .h file with the preprocessed data being a type of const uint8_t. In the second option, to create an .h file the process is to print, with the print_int, the two keys and copy those data from the terminal (see Chapter 4.1.2). The last step is to paste the keys to the .h file.

6.7 RamtoFlash -Static arrays - .h files - Encryption - AES640

The function of encryption has many variables and needs space up to 51248B. Except that, the function has an extra function inside which needs space up to 30784B, meaning that the program requires 82032B to run without the public key and ciphertext. The ciphertext uses space up to 9720B in SRAM and the public key uses 9616B. Due to those variables, the space the program needs to run is 101368B. As a result, a microcontroller with low SRAM cannot afford to run the encryption**()**.

CODE	CODE CHANGES	THEORY
uint8_t *pk_seedA=&pk[0]	*pk_seedA=&pk[0](FLASH)	seedA
uint8_t *pk_b=&pk[16]	*pk_b=&pk[16](FLASH)	B
uint8_t ct[9732]	uint8_t ct[9732]	Ciphertext
uint8_t *ct_c1=&ct[0];	uint8_t *ct_c1=&ct[0];	B’
uint8_t *ct_c2=&ct[ ]	uint8_t *ct_c2=&ct[ ]	C
uint16_t B[5120]	uint16_t *B=&Sp[0]	B’
uint16_t V[64]	uint16_t V[64]	V
uint16_t C[64]	uint16_t C[64]	C
uint16_t Bp[5120]	uint16_t *Bp=&Sp[0]	
uint16_t Sp[10304]	uint16_t Sp[10304] send the data in Flashuse it as Spdata	S
uint16_t *Ep=&Sp[5120]	uint16_t *Ep=&Sp[5120] send the data in Flash read it uint16_t *Ep=&Spdata[5120]	E’
uint16_t *Epp=&Sp[10240]	uint16_t *Epp=&Sp[10240] send the data in Flash,read it as uint16_t *Epp=&Spdata[10240]	E’’
uint8_t G2in[32]	uint8_t G2in[32]	
uint8_t *pkh=&G2in[0]	uint8_t *pkh=&G2in[0]	
uint8_t *mu=&G2in[16]	uint8_t *mu=&G2in[16]	
uint8_t G2out[32]	uint8_t G2out[32]	
uint8_t *seeds=&G2out[0]	uint8_t *seeds=&G2out[0]	
uint8_t *k=G2out[16]	uint8_t *k=G2out[16]	
uint8_t Fin[9748]	uint8_t *Fin=&Sp[0]	
uint8_t *Fin_ct=&ct[0]	uint8_t *Fin_ct=&Sp[0]	
uint8_t *Fin_k=&ct[9732]	uint8_t *Fin_k=&Sp[4860]	
uint8_tSHAKE_input_seedSE[17]	uint8_tSHAKE_input_seedSE[17]	

Table 9: Variables in code and Variables in theory Encaps()

To manage the problem certain techniques were developed in this thesis. First, the public key is in the .h file, meaning that it is stored in Flash. After, the function has many variables which use space up to 51248B. The variables are B and Bp. each of which needs 10240B. The variables V and C which use 128B, G2in and G2out which require 32B, SHAKE_input_seedSE which needs 17B, Fin which requires 9736B, and Sp which requires 20608B.

Another technique to reduce space is to delete the variables B, Bp, and Fin and replace them with Sp as a static array. First, Sp is generated and data is normalized with the function sample_n. The last step is to keep Sp in flash with the usage of RamtoFlash and after create a point named Spdata points in a specific address in Flash.

```
RamtoFlash(Sp,(2 * PARAMS_N + PARAMS_NBAR) * PARAMS_NBAR,ADDR_FLASH_PAGE_40

uint32_t Address = ADDR_FLASH_PAGE_40;
uint16_t *Spdata;
Spdata=(__IO uint64_t*)Address;

uint16_t *Ep = &Spdata[PARAMS_N * PARAMS_NBAR];              // contains s
uint16_t *Epp = &Spdata[2 * PARAMS_N * PARAMS_NBAR];         // contains s
```

Figure 20: Send the data of Sp in the Flash and create a pointer in Flash to read the data with a pointer

As described above, Sp is a static array due to having various uses. After the generation and transmission of data in Flash, Sp takes the place of Bp with the function mul_add_sa_plus_eand there Bp is computed, which in this case is Sp. After Sp is packed in ct_c1 which is a pointer for the ciphertext. The next step for Sp is to take the place of B and to partake in two functions, namely unpacking the public key therein and in computing V=Spdata*Sp(B)+Epp with mul_add_sb_plus_e and after the second pointer of the ciphertext is generated through ct_c2.

The last step inside the function of encryption is to create two uint8_t pointers, Fin_ct and Fin_K, to point to different places of Sp. The first point is *Fin_ct=&Sp[0] and the second is *Fin_K=&Sp[4860].

```
uint8_t *Fin_ct = &Sp[0];
uint8_t *Fin_k = &Sp[4860];
// Compute ss = F(ct||KK)
```

Figure 21: Create a pointer uint8_t and point the pointer in an uint16_t array.

6.8 RamtoFlash - Static arrays - .h files - Decryption - AES640

CODE	CHANGE CODE	THEORY
uint16_t B[5120]		A
uint16_t Bp[5120]	uint16_t *Bp = &Ep[0]; send the data in Flashread it as uint16_t *Bpdata= (__IOuint64_t*)Address;	B’
uint16_t W[64]	uint16_t W[64]	V
uint16_t C[64]	uint16_t C[64]	C
uint16_t CC[64]	uint16_t CC[64]	C’
uint16_t BBp[5120]	uint16_t *Bp = &Ep[0];	B’’
uint16_t Sp[10304]	uint16_t Sp[10304] send the data in FLASH and read it as uint16_t *Spdata= (__IOuint64_t*)Address;	S
uint16_t *Ep=&Sp[5120]	uint16_t *Ep=&Sp[5120] send the data in Flash read it uint16_t *Ep=&Spdata[5120]	E’
uint16_t *Epp=Sp[10240]	uint16_t *Epp=&Sp[10240] send the data in Flash,read it as uint16_t *Epp=&Spdata[10240]	E’’
uint8_t *ct_c1=&ct[0]	uint8_t *ct_c1=&ct[0] (FLASH)	B’
uint8_t *ct_c2=&ct[ ]	uint8_t *ct_c2=&ct[ ](FLASH)	C
uint8_t *sk_s=sk[0]	uint8_t *sk_s=sk[0] (FLASH)	
uint8_t *sk_pk=&sk[16]	uint8_t *sk_pk=&sk[16](FLASH)	seedA
uint8_t *sk_S=&sk[9632]	uint8_t *sk_S=&sk[9632](FLASH)	S
uint16_t S[5120]	uint16_t *S = &Sp[0];	
uint8_t *sk_pkh=sk[19872]	uint8_t *sk_pkh=sk[19872]	
uint8_t *pk-_seedA=&sk[16]	uint8_t *pk_seedA=&sk[16]	seedA
uint8_t *pk_b=&sk[32]	uint8_t *pk_b=&sk[32]	B
uint8_t G2in[32]	uint8_t G2in[32]	
uint8_t *pkh=&G2in[0]	uint8_t *pkh=&G2in[0]	
uint8_t *muprime=&G2in[16]	uint8_t *muprime=&G2in[16]	
uint8_t G2out[32]	uint8_t G2out[32]	
uint8_t &seedSEprime=&G2out[0]	uint8_t &seedSEprime=&G2out[0]	
uint8_t *kprime=&G2out[16]	uint8_t *kprime=&G2out[16]	
uint8_t Fin[9748]	uint8_t *Fin=&Sp[0]	
uint8_t *Fin_ct=&Fin[0]	uint8_t *Fin_ct=&Sp[0]	
uint8_t *Fin_k=Fin[9732]	uint8_t *Fin_k=&Sp[4860]	
uint8_t *SHAKE_input_seedSEprime[17]	uint8_t *SHAKE_input_seedSEprime[17]	

Table 10: Variables in code and Variables in theory Decaps()

The Decryption function has a lot of variables and takes up a lot of space, up to 71896B. Except for the fact that the function contains an extra function which create the matrix A (4.4 Generating Matrix A)that takes up to 30784B of memory, the program requires 102680B of memory to run without the secret key and ciphertext. Ciphertext uses 9720B RAM and the key 19888B.All the variables need space 132288B.

The implementation to reduce the space begins with the secret key and ciphertext are initially stored in the .h file, indicating that they are in Flash. B, Bp, and BBp variables require 10240B; W, CC, C require 128B; G2in, G2out require 32B; SHAKE input seed_SE variables demand 17B; Fin requires 9736B, and Sp requires 20608B.

Another way to save space was to remove some variables (B, BBp, and Fin) and replace them with a static array (Sp). First, the ciphertext pointers ct_c1 and ct_c2 used the function of unpacking to transfer their content to Bp and C. To maintain the Bp data, the function of RamtoFlash was used and data was transferred to Flash, then the pointer was referred to with the name Bpdata and used instead of Bp.

```
RamtoFlash(Bp, PARAMS_N  * PARAMS_NBAR,ADDR_FLASH_PAGE_70,ADDR_FLASH_PAGE_83);

uint32_t Address = ADDR_FLASH_PAGE_70;
uint16_t *Bpdata;
Bpdata=(__IO uint64_t*)Address;

// Generate (seedSE' || k') = G_2(pkh || mu')
memcpy(pkh, sk_pkh, BYTES_PKHASH);
shake(G2out, CRYPTO_BYTES + CRYPTO_BYTES, G2in, BYTES_PKHASH + BYTES_MU);
```

Figure 22: Send the data of Bp in the Flash and create a pointer in Flash to read the data

Because of its multiple uses, Bp is a static array, as stated in the previous paragraph. After the production and transmission of data to Flash, Bp took the position of BBp in the function mul- add sa plus e, where Bp is computed, and then the Bp module of q, which is a parameter, was reduced. The next step was to use the function of ct to verify the results of BBp and Bp with the data of BBp being located at Bp and the data of Bp at Bpdata in Flash and return an integer to selector1.

The next stage is to compute the data of B, and in place of Bthere is Bp, which uses data from the public key to calculate W=Sp*B+Epp, using the function of unpack. Following those procedures, CC is computed and C and CC are verified using the function ct_verify, which returns an integer int8_t with the name selector2. After that, the selector1| selector2 result has to be checked and saved in the variable int8_t selector.

The final stage in the function of encryption is to generate two uint8_t pointers, Fin ct and Fin K, which point to distinct locations of Sp, namely *Fin ct=&Sp[0] and *Fin K=&Sp[4860].

```
uint8_t *Fin_ct = &Sp[0];
uint8_t *Fin_k = &Sp[4860];
// Compute ss = F(ct||KK)
```

Figure 23: Create a pointer uint8_t and point the pointer in an uint16_t array

The aforementioned approach used for the encryption, decryption, and Keypair stages was repeated for the implementation of FrodoKEM640-SHAKE where the released memory was 303472 for encryption, 46928 for decryption, and10218 for Keypair. As for encryption and decryption, the keys were in an .h file.

6.9 Optimization Techniques

There are two types of optimizations the -Ofast and -O0. Optimization for speed was chosen using -Ofast to improve performance. More specifically, Arm Compiler makes us of certain optimizations to boost application performance. Based on the user's goal, the relevant optimization is chosen. As regards optimizing performance, the recommended optimizations are -O2, -O3, and -Omax, along with -Ofast [25].-Omax is not an available option for STM32L552 and STM32WL55JC1. Therefore, all the rest optimizations were checked, and the final optimization which was chosen was -Ofast as it was considerably faster. -Ofast runs optimizations from level -O3 and involves the optimizations that run with the option of -ffast-math armclang option. -Ofast runs aggressive optimizations, too, which may cause language compliance violations, negatively affect debugging, and increase the size of the code in comparison with -O3.Choosing the level -O0 (default) of the command line option -On, allows for no optimization. [25] That is why it is common practice to utilize -O0 for debugging.

7. Results

This chapter reports the results of the research. These results refer to the two microcontrollers STM32WL55JC1 and STM32L552ZE and more specifically to the implementation time. Then a feature work is reported which analyses one of the ways that FrodoKEM could operate in an IoT system.

7.1 Results for TIMER for STM32L552 without -Ofast optimization

The STM32L552 implements ten different projects for FrodoKEM. The first five conclude the AES and the other five only the SHAKE128. The table shows the time that the microcontroller needs to run each function. The time is counted with TIMER2.The results above in the table present the time which is required each time for every function to run. Furthermore, the results show that the form which two-way cache works more effectively for this thesis program. A remarkable option is that the program with usage of flash with two-way cache has similar results as the one-way cache without usage of flash. The FrodoKEM640 with the usage of SHAKE128 to generate matrix A shows that the algorithm of only using SHAKE128 is quicker than FrodoKEM640 which uses AES-128. The results present that it is more effective to use one-way cache forSHAKE128.

FRODOKEM	KEYPAIR	ENCRYPTION	DECRYPTION
AES 1-WAY CACHE	10,386928s	10,819695s	10,800936s
AES 2-WAY CACHE	9,240595s	9,684542s	9,656033s
AES WITHOUT CACHE	15,902839s	16,502325s	16,483416s
AES 1-WAY CACHE & FLASH	11,195637s	11,649415s	11,806252s
AES 2-WAY CACHE & FLASH	10,046823s	10,505343s	10,663041s
SHAKE 1-WAY CACHE	6,296684s	7,978692s	7,959624s
SHAKE 2-WAY CACHE	7,23780s	8,94332s	8,91431s
SHAKE 1-WAY CACHE & FLASH	7,126521s	8,824282s	8,995053s
SHAKE 2-WAY CACHE & FLASH	8,06187s	9,79278s	9,95312s
SHAKE WITHOUT CACHE	9,244770s	12,691196s	12,676077s

Table 11: FrodoKEM640 AES and SHAKE results for STM32L552 from timer

FRODOKEM	KEYPAIR	ENCRYPTION	DECRYPTION
AES 1-WAY CACHE	10,480978918s	10,917535s	10,898455s
AES 2-WAY CACHE	9,325938s	9,703519s	9,7453551s
AES WITHOUT CACHE	15,871298318s	16,614699s	16,600649s
AES 1-WAY CACHE & FLASH	11,297804s	11,755729s	11,914011s
AES 2-WAY CACHE & FLASH	10,109130s	10,601200s	10,761207s
SHAKE 1-WAY CACHE	6,254449s	8,051011s	8,032435s
SHAKE 2-WAY CACHE	7,245729s	8,951207s	8,928169s
SHAKE 1-WAY CACHE & FLASH	7,191769s	8,880082s	8,999093s
SHAKE 2-WAY CACHE & FLASH	8,063111s	9,802377s	9,963410s
SHAKE WITHOUT CACHE	9,328562s	12.806221s	12,790996s

Table 12: FrodoKEM640 AES and SHAKE results for STM32L552 from debugger

7.2 Results for TIMER for STM32L552 with -Ofast optimization

The STM32L552 implements ten different projects for FrodoKEM. The first five conclude the AES and the other five only the SHAKE128. The table show the time that the microcontroller needs to run each function. The time is counted with TIMER2.There is no actual difference as there are almost identical results in the run-time of cache, either when it comes to one-way or two-way cache, when having introduced -Ofast optimization for STM23L552. When cache is not used, there is a considerable increase in time of almost 2 seconds.

When using SHAKE128 to produce matrix A, the results of FrodoKEM640 present that by only exploiting SHAKE128 the algorithm becomes faster than that of FrodoKEM640 which only involves AES-128. Therefore, it is shown that one-way cache for SHAKE128 is more efficient. Comparing the above data from 7.1 and 7.2, it is evident that there is less runtime using -Ofast compared to using -O0 (default) and performance improves, as well.

FRODOKEM	KEYPAIR	ENCRYPTION	DECRYPTION
AES 1-WAY CACHE	1,495845s	1,522546s	1,514929s
AES 2-WAY CACHE	1,494458s	1,521095s	1,513586s
AES WITHOUT	3,844165s	3,916926s	3,903146s

CACHE			
AES 1-WAY CACHE & FLASH	2,317678s	2,334294s	1,932913s
AES 2-WAY CACHE & FLASH	2,316002s	2,331745s	2,506324s
SHAKE 1-WAY CACHE	1,309897s	1,505160s	1,497324s
SHAKE 2-WAY CACHE	1,296647s	1,491468s	1,483770s
SHAKE 1-WAY CACHE & FLASH	2,123165s	2,293968s	2,480437s
SHAKE 2-WAY CACHE & FLASH	2,115956s	2,286825s	2,485735s
SHAKE WITHOUT CACHE	2,108932s	2,266745s	2,457345s

Table 13: FrodoKEM640-AES and SHAKE with -Ofast

7.3 Results for TIMER for STM32WL55JC1 with -Ofast optimization

The FrodoKEM640 AES at the STM32WL55JC1 is less fast. The difference is that this microcontroller has the AES accelerator which can reduce the time of run to a significant level.On the other hand, the FrodoKEM640 SHAKE is an algorithm which uses less time to implement in comparison with FrodoKEM640 AES.

	KEYPAIR	**ENCRYPTION**	**DECRYPTION**
FrodoKEMAES	4,871132s	6,055257s	6,392991s
FrodoKEMSHAKE	5,092828s	5,617887s	6,879387s

Table 14: FrodoKEM640AESSHAKE results for STM32WL55

7.4 Final remarks

Nowadays, security on embedded devices is in an emergency. New post-quantum computers can attack and take access to private data. The main goal of this thesis is to apply a post quantum algorithm as FrodoKEM to two different microcontrollers with low MHz in comparison with a PC processor. That is why FrodoKEM can be used in IoT programs or for signatures to keep data safe from vulnerable attacks as a valuable tool for the increasing needs of the modern IT industry.

7.5 Conclusion

For the needs of this thesis, the algorithm FrodoKEM was used to improve the security of smaller-sized and embedded devices taking into consideration the rise of cryptanalytic attacks owing to the ongoing developments of quantum computing. The encapsulation mechanism of FrodoKEM was tested using a two-way cache on the

STM32L552 microcontroller, which enabled the algorithm to run faster with AES. Results showed that the SHAKE128 algorithm ran faster on the STM32L552 microcontroller compared to the lower speed resulting from the implementation of FrodoKEM. Finally, the SRAM space in STM32WL55JC1 was minimized after data transmission was directed to Flash memory, which combined with the aforementioned tests proved to optimize security. Based on the adaptability of WLSSJC1 which allows for the connection to a wireless LoRa Network and Cloud computing, in general, the findings of the applied FrodoKEM scheme indicate that the implementation of this algorithm is applicable and effective for the security, management, and overall performance of IoT systems.

7.6 Future work

Nowadays, the rapid increase of the IoT systems is a fact. Many researchers upgrade their knowledge and they find new technologies to make the lives of people more efficient. The main problem with new technologies is that it is necessary to find new solutions and technologies to keep data safe. Future research ought to be conducted about an IoT system which is protected by a post-quantum algorithm. The main requirement is that it is important for the boards to send their data to a trusted device as a Cloud. Moreover, a plan to materialize this is to request access from the Cloud again whenever a microcontroller goes through some firmware update or a reset. In Figure 24, there is Bob (i.e., user) and the Cloud system. Every time an embedded system needs to reset or do a firmware update, it starts encryption with the public key of the Cloud with the usage of FrodoKEM and is sent for verification. If all is right, the microcontroller gains access to the Cloud again. The connectability of WLSSJC1 with a LoRaWAN network featuring secure edge gateways is able to implement a FrodoKEM algorithm to boost the overall performance of an IoT system [5].

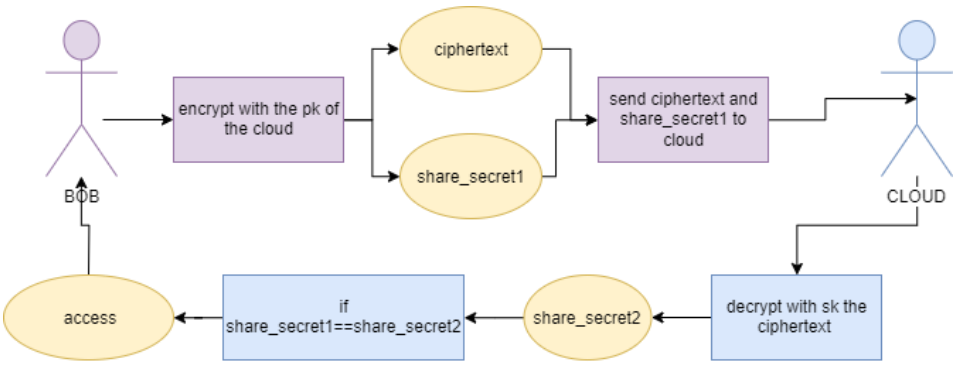


Figure 24: Bob implements the encryption with the public key of the Cloud system and the Cloud system the decryption with secret key to check if there is an attack

References

1. [FIPS]. (2001, november 26). Announcing the Advanced Encryption Standard (AES).

2. Alkim, Boss, Ducas, Longa, Mironov, Naehrig, et al. (2021). *FrodoKEM: Learning with Errors*. Retrieved from https: //FrodoKEM.org/files/FrodoKEM-specification-20210604.pdf.

3. An, SangWoo, and Seog Chung Seo. (2020). "Efficient Parallel Implementations of LWE-Based Post-Quantum Cryptosystems on Graphics Processing Units" *Mathematics* 8, no. 10: 1781. https://doi.org/10.3390/math8101781

4. Asif, Rameez. 2021. "Post-Quantum Cryptosystems for Internet-of-Things: A Survey on Lattice-Based Algorithms" *IoT* 2, no. 1: 71-91. https://doi.org/10.3390/iot2010005

5. Bhasin, Shivam, Jan-Pieter D'Anvers, Daniel Heinz, Thomas Pöppelmann and Michiel Van Beirendonck. "Attacking and Defending Masked Polynomial Comparison for Lattice-Based Cryptography." *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2021 (2021): 334-359.

6. Marcello Coppola and George Kornaros, "Automation for Industry 4.0 by using Secure LoRaWAN Edge Gateways", in L. Andrade, F. Rousseau, (eds), Multi-Processor System-on-Chip, vol. 2., *ISTE Ltd, London, and Wiley, New York, March 2021*, https://iste.co.uk/book.php?id=1739 , ISBN : 9781789450224

7. FIPS. (2015). Announcing the SHA-3 .

8. Fritzmann, Tim, Jonas Vith and Martha Johanna Sepúlveda. "Strengthening Post-Quantum Security for Automotive Systems." *2020 23rd Euromicro Conference on Digital System Design (DSD)* (2020): 570-576.

9. Howe, James, Tobias Oder, Markus Krausz and Tim Güneysu. "Standard Lattice-Based Key Encapsulation on Embedded Devices." *IACR Cryptol. ePrint Arch.* 2018 (2018): 686.

10. Huber, & Herrmann. (n.d.). (2021)A Long-term Security Concept for IoT Products Dr. Hans Herrmann Head of Embedded Systems at cogitron GmbH Pliening near Munich., *EmbeddedWORLD2021*

11. JIHOON, BO-YEON, JOOHEE, IL-JU, & TAE-HO. (2020). Single-Trace Attacks on Message Encoding in. *IEEE Access.* vol. 8, pp. 183175-183191, 2020, doi: 10.1109/ACCESS.2020.3029521.

12. Khalid, Ayesha, Sarah McCarthy, Weiqiang Liu and Máire O'Neill. "Lattice-based Cryptography for IoT in A Quantum World: Are We Ready?" *2019 IEEE 8th*

International Workshop on Advances in Sensors and Interfaces (IWASI) (2019): 194-199.

13. George Kornaros, Ioannis Christoforakis, Othon Tomoutzoglou, Dimitrios Bakoyiannis, Kallia Vazakopoulou, Miltos Grammatikakis, Antonis Papagrigoriou, “Hardware Support for Cost-Effective System-Level Protection in Multi-core SoCs.”, *2015 Euromicro Conference on Digital System Design*, DSD 2015, Madeira, Portugal, August 26-28, 2015, pp. 41-48, DOI: 10.1109/DSD.2015.65

14. G. Kornaros, O. Tomoutzoglou, D. Mbakoyiannis, N. Karadimitriou, M. Coppola, E. Montanari, I. Deligiannis, G. Gherardi, “Towards Holistic Secure Networking in Connected Vehicles through Securing CAN-bus Communication and Firmware-over-the-Air Updating”, *Journal of Systems Architecture* (2020), vol. 109, pp. 101761, ISSN 1383-7621, doi: https://doi.org/10.1016/j.sysarc.2020.101761

15. S. Leivadaros, G. Kornaros and M. Coppola, “Secure Asset Tracking in Manufacturing through Employing IOTA Distributed Ledger Technology”, i*n The 21th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID’21), 2nd Workshop on Secure IoT, Edge and Cloud systems (SIoTEC),* May 10-13, 2021

16. ST life.augmented, S. (2022). STM32WL55xx STM32WL54xx: Multiprotocol LPWAN .

17. Mironov, E. A. (2021, june 4). FrodoKEM Learning With Errors Key Encapsulation. p. 59.

18. Regev, Oded. “On lattices, learning with errors, random linear codes, and cryptography.” *STOC '05* (2005).

19. O. Regev, "The Learning with Errors Problem (Invited Survey)," *2010 IEEE 25th Annual Conference on Computational Complexity*, 2010, pp. 191-204, doi: 10.1109/CCC.2010.26.

20. Semantic, Scholar, Joppe, Friedberger, Martinoli, Oswald, et al. (2018). Fly, you fool! Faster Frodo for the ARM Cortex-M4. *Cryptology ePrint Archive,* Paper 2018/1116. https://eprint.iacr.org/2018/1116

21. Shahbazi, Karim and Seok-Bum Ko. “Area and power efficient post-quantum cryptosystem for IoT resource-constrained devices.” *Microprocess. Microsystems* 84 (2021): 104280.

22. ST life.augmented. (2020). STM32L552xx: Ultra-low-power Arm® Cortex®-M33 32-bit .

23. Thomas, Qian, & Alexander. (n.d.).(2020) A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. *Cryptology ePrint Archive*, Paper 2020/743 https://eprint.iacr.org/2020/743

24. G. Trouli and G. Kornaros, "Automotive Virtual In-sensor Analytics for Securing Vehicular Communication," *in IEEE Design & Test*, vol.37, issue 3, pp. 91-98, print

ISSN: 2168-2356, online ISSN: 2168-2364, https://ieeexplore.ieee.org/document/9001022, June 2020, **DOI: 10.1109/MDAT.2020.2974914**

25. Selecting optimization options. (2022). Retrieved from https: //developer.arm.com/documentation/100748/0612/using-common-compiler-options/selecting-optimization-options

26. Boldyreva, A., Goyal, V., & Kumar, V. (2008). Identity-based Encryption with Efficient Revocation. Retrieved from https://faculty.cc.gatech.edu/~aboldyre/papers/bgk.pdf

27. Minelli, M. (2018). Fully homomorphic encryption for machine learning. Cryptography and Security [cs.CR]. Université Paris sciences et lettres. Retrieved from https://tel.archives-ouvertes.fr/tel-01918263/document
28. FrodoKEM. (n.d.). Retrieved from https://www.microsoft.com/en-us/research/project/frodokem/
29. Lindner, R., & Peiker, C. (2011). Better Key Sizes (and Attacks) for
30. LWE-Based Encryption. In A. Kiayias (Ed.): CT-RSA 2011, LNCS 6558 (pp. 319–339). Springer: Sans Fransisco, California.