Module-lattice KEM Over a Ring of Dimension 128 for Embedded Systems

François Gérard

Université libre de Bruxelles
`fragerar@ulb.ac.be`

Abstract. Following the development of quantum computing, the demand for post-quantum alternatives to current cryptosystems has firmly increased recently. The main disadvantage of those schemes is the amount of resources needed to implement them in comparison to their classical counterpart. In conjunction with the growth of the Internet of Things, it is crucial to know if post-quantum algorithms can evolve in constraint environments without incurring an unacceptable performance penalty. In this paper, we propose an instantiation of a module-lattice-based KEM working over a ring of dimension 128 using a limited amount of memory at runtime. It can be seen as a lightweight version of Kyber [7] or a module version of Frodo [8]. We propose parameters targeting popular 8-bit AVR microcontrollers and security level 1 of NIST. Our implementation fits in around 2 KB of RAM while still providing reasonable efficiency and 128 bits of security, but at the cost of a reduced correctness.

1 Introduction

Since the publication of an efficient quantum algorithm able to break both discrete logarithm and factorization problems [25], a joint effort has been made from part of the cryptographic community to overcome this future threat under the name of post-quantum cryptography. Recently, the National Institute of Standards and Technology asked researchers to develop and implement several proposals for post-quantum public-key algorithms to pave the way to standardization. After the first round of submissions, the most represented family of post-quantum scheme is the one using lattices problems. For several years, lattice-based cryptography has seen a lot of development, in theory, as well as in practice. Lattice-based problems like learning with errors (LWE) [24] offer a huge versatility in terms of cryptographic applications, ranging from the most basic primitives like signatures and key-exchanges, to advanced constructions such as fully homomorphic encryption. During its early phase, it was unclear if this problem was enabling practical instantiations since it required to work with huge matrices over the integers. Latter, a more structured version of the problem based on operations in the ring of integers of number fields was proposed under the name ring learning with errors (RLWE) [21]. It reduced drastically keys sizes and sped up computation thanks to efficient polynomial algorithms based on a discrete Fourier transform. Unfortunately, it also reduced the confidence we have in these cryptosystems since security is now based on problems in algebraically structured lattices called ideal lattices. Recently, a new approach based on module lattices has been successfully used to take the middle ground between LWE and RLWE, while maintaining efficiency. Nevertheless, even if proposals to the NIST competition were packaged with optimized software, the primitives and parameters choices stayed generic and were targeting modern personal computers with common architecture such as x86 CPUs. In the real world, cryptography is also used to secure devices found in the myriad of embedded systems appearing with the internet of things (IoT). Those devices live in a really constraint environment, often with low computing power and very limited storage. While slow primitives can be acceptable if no firm time constraints have to be met or if cryptography is only a marginal part of the whole system, the lack of memory space to run those primitive is a real issue in practice and is not fixable without modifying the hardware. Let us, for example, take NewHope[3] [2], it is defined as a Diffie-Hellman like key exchange over the ring $\mathbb{Z}_{12289}[X]/\langle X^{1024}+1\rangle$ which means that, even using an optimal encoding, one ring element takes over 1700 KB of RAM. On small embedded devices such has the ATmega328p, it means that one polynomial already consume the whole memory. [1] Hence, it is interesting to study what kind of trade-offs can be made in order to bring post-quantum algorithm to the embedded world. Naturally, some sacrifices will be made in security, correctness or efficiency to obtain a lightweight version of those post-quantum primitives but small devices often have a threat model different than a general computer connected to the internet and it is not unusual to lighten cryptographic schemes in this context.

Our work. In this work, we propose an instantiation of a post-quantum key encapsulation mechanism (KEM) based on the hardness of problems on module lattices with parameters tailored for low memory devices. The scheme can be seen as a module version of Frodo or a lightweight, uncompressed, version of Kyber. We provide experimental results obtained with a software implementation on 8-bit AVR microcontrollers showing that it is indeed possible to implement a lattice-based key exchange on very constraint platforms without scarifying too much security. The novelty in comparison to Kyber is the usage of the ring $\mathbb{Z}_q[X]/\langle X^{128}+1\rangle$ offering more versatility in the choice of the parameters and more specifically allowing to set q to 257 while still benefiting from the discrete Fourier transform for polynomial multiplication. Having a ring of smaller dimension also offers other advantages such as a less algebraically structured problem and limits the amount of coefficients unpacked during the execution of the algorithm. We maintain a classical security level of more than 128 bits and a post-quantum security of around 100 bits with pessimistic analysis. The main drawback is the lowered correctness preventing those parameters to be used in full generality.

Previous work. Lattice-based cryptography implementations on low-power devices has been studied under different angles in the litterature [13,4,22,11,6,9]. Implementations specifically targeting 8-bit AVR microcontrollers can be found in [23,20,19]. Since standard LWE is to heavy for embedded systems, they focus on RLWE. Up to our knowledge, the specific case of MLWE has not yet been treated extensively in the literature. The main reason it that it is a pretty new construction and most of the issues are common to RLWE since the bottleneck of computation is arithmetic in polynomial rings.

[1] Obviously, NewHope was not made to fit specifically on small devices, it offers really high security and was crafted as a general key exchange targeting future standardization.

Software The software written for this work is placed in the public domain and is available at `https://github.com/fragerar/Module-KEM-128`

2 Preliminaries

2.1 Notations

We use $\mathbb{Z}_q$ to denote the ring of integers modulo q and $\mathcal{R}$ for the polynomial ring $\mathbb{Z}_q[X]/\langle X^n + 1\rangle$. Elements of $\mathcal{R}$ and scalars are represented by lower case letters (e.g. $p \in \mathcal{R}$). For vectors, we use bold lower case letters ($\mathbf{v} \in \mathcal{R}^k$) and for matrices, bold capital letters ($\mathbf{M} \in \mathcal{R}^{k\times k}$). An algorithm $\texttt{alg}$ ran on input x with result put in y is written $y \leftarrow \texttt{alg}(x)$ or $y = \texttt{alg}(x)$ if it is in an equation. Sometimes we want to explicitly provide a random tape r to $\texttt{alg}$ and write $y \leftarrow \texttt{alg}(x; r)$. The operation of sampling a value v from a distribution χ is noted $v \overset{r}{\leftarrow} \chi$. When a set is provided instead of an explicit distribution, we mean the uniform distribution over this set. Since we work with the Number Theoretic Transform, we may want to explicitly indicate whether a polynomial is in frequency domain. In this case, we use the tilde notation: $\tilde{v} \leftarrow \texttt{NTT}(v)$. For a vector of polynomials $\mathbf{v}$, we write $\|\mathbf{v}\|_\infty$ the maximum of the set of the absolute values of the coefficients of all its entries.

2.2 Cryptographic notions

Sharing a key in order to communicate over an insecure channel using symmetric algorithm is one of the most fundamental task of public-key cryptography. This can be achieved using a key encapsulation mechanism (KEM) constructed from a public-key encryption scheme (PKE) semantically secure against chosen plaintext attacks (CPA). The transition from a CPA-secure PKE to a CPA-secure KEM is straightforward, the key is sampled at random and encrypted under the public key of the recipient who retrieve it using his private key. Obtaining a KEM secure against chosen ciphertext attacks (CCA) is more complex and require special transformations such as Fujisaki-Okamoto [12]. In the following we review basic notions of CPA-PKE and CCA-KEM. All algorithms implicitly take as input the public parameters of the system and the security parameter.

Public-key encryption scheme

Definition 1. (PKE). A public-key encryption scheme (PKE) is a tuple ($\texttt{KeyGen}$, $\texttt{Encrypt}$, $\texttt{Decrypt}$) composed of the following algorithms :

– $\texttt{KeyGen}()$: a randomized algorithm outputing a secret/public key pair (sk,pk).
– $\texttt{Encrypt}(pk, m)$: a randomized algorithm taking as input a public key pk and a message m and outputing a ciphertext ct
– $\texttt{Decrypt}(sk, ct)$: a deterministic algorithm taking as input a secret key sk and a ciphertext ct and outputing a message m.

Definition 2. (**CPA security for PKE**) A PKE is said to be CPA-secure if the probability that any polynomial time adversary (modeled as a set of two algorithms $\{\mathcal{A}, \mathcal{A}'\}$ is negligibly close to $\frac{1}{2}$:

– The challenger runs the key generation algorithm and outputs a key pair $(sk, pk) \leftarrow \texttt{Keygen}()$.

– The adversary learns the pk and outputs two messages $(m_0, m_1) \leftarrow \mathcal{A}(pk)$.
– The challengers choose a bit b and outputs a ciphertext $ct \leftarrow \texttt{Encrypt}(pk, m_b)$.
– The adversary outputs a bit $b' \leftarrow \mathcal{A}'(ct, pk)$ and wins the game if $b = b'$

Key encapsulation mechanism

Definition 3. (KEM). A key encapsulation mechanism (KEM) with key space $\mathcal{K}$ is a tuple (`KeyGen`, `Encaps`, `Decaps`) composed of the following algorithms :

– `KeyGen`(): a randomized algorithm outputing a secret/public key pair (sk,pk).
– `Encaps`(pk): a possibly randomized algorithm taking as input a public key and ouputing a ciphertext ct, together with a symmetric key $K \in \mathcal{K}$.
– `Decaps`(sk, ct): a deterministic algorithm taking as input a secret key and a ciphertext and outputing a symmetric key K'.

Definition 4. (δ-Correctness) [14] A key encapsulation mechanism is said to be δ-correct if

$$\mathbb{P}[\texttt{Decaps}(sk, ct) \neq K' \mid (sk, pk) \leftarrow \texttt{KeyGen}(); (ct, K) \leftarrow \texttt{Encaps}(pk)] \leq \delta$$

It captures the fact that the value shared by the two parties is indeed the same with probability $1 - \delta$.

Definition 5. (**CCA security for KEM**) A KEM is said to be CCA-secure if the probability that any polynomial time adversary (modeled as an algorithm $\mathcal{A}$) is negligibly close to $\frac{1}{2}$:

– The challenger runs the key generation algorithm and outputs a key pair $(sk, pk) \leftarrow \texttt{Keygen}()$.
– The challenger gets a ciphertext/key pair by running $(ct, K_0) \leftarrow \texttt{Encaps}(pk)$ and generates a random key $K_1 \in \mathcal{K}$.
– The challengers choose a bit b and reveal k_b to the adversary.
– The adversary outputs a bit $b' \leftarrow \mathcal{A}(ct, k_n, pk)$. Here, $\mathcal{A}$ has access to a decapsulation oracle $\mathcal{O}^{decaps}$ defined as $\texttt{Decaps}(sk, \cdot)$. The only restriction is that it will not accept queries on ct. The adversary wins the game if $b = b'$.

2.3 Module learning with errors

We base the hardness of our scheme on the module learning with errors (MLWE) problem. It has been extensively studied by Langlois and Stehlé in [17]. Let $\mathcal{R} = \mathbb{Z}_q[X]/\langle X^n + 1\rangle$ and χ a narrow error distribution over $\mathcal{R}$, that is to say outputting polynomials with small coefficients over the integers with high probability. Its decision version is defined as follow :

Definition 6. ((Decisional-)MLWE). Let $\mathbf{a}_i \overset{r}{\leftarrow} \mathcal{R}^k$. For a secret vector $\mathbf{s} \leftarrow \mathcal{R}^k$ and a polynomially bounded number of samples $\mathbf{t}_i = \langle \mathbf{a}_i, \mathbf{s}\rangle + e_i \in \mathcal{R}$ with $e_i \overset{r}{\leftarrow} \chi$, distinguish the distribution of the $\mathbf{t}_i$ from the uniform distribution over $\mathcal{R}$. For a fixed number of samples m, we write the problem in matrix form where, for $\mathbf{A} \in \mathcal{R}^{m\times k}$, the goal is to distinguish $\mathbf{A} \cdot \mathbf{s} + \mathbf{e}$ from the uniform distribution over $\mathcal{R}^m$

If one set $n = 1$, the MLWE problem is simply the LWE problem [24]. If one set $m = k = 1$, it becomes the RLWE problem [21].
It is also possible to consider multiple instances of MLWE at the same time where the secret is now a matrix $\mathbf{S}$ over $\mathcal{R}^{k \times m'}$. As shown in [17], this problem enjoys worst-case to average-case reductions from lattices problems believed to be hard for a quantum adversary.

2.4 Binomial distribution

As in [7,3], we use the centered binomial as error distribution in order to facilitate sampling. Since the support of the error distribution can be taken quite small thanks to our choice of $q = 257$, an approach based on gaussian sampling with precomputed table is also possible.
The center binomial of parameter p, denoted $\mathcal{B}_p$ is the distribution of the outcome of $\sum_{i=1}^{p}(b_i - b'_i)$ with b_i and b'_i sampled uniformly at random in $\{0, 1\}$. For our modulus q, we only need to use $\mathcal{B}_1$ to get secure MLWE instances. Hence, the distribution is simply: $\mathbb{P}[X = -1] = 0.25$, $\mathbb{P}[X = 0] = 0.5$ and $\mathbb{P}[X = 1] = 0.25$.

2.5 Number Theoretic Transform

When the ring $\mathcal{R} = \mathbb{Z}_q[X]/\langle X^n + 1\rangle$ is instanciated with n a power of 2, q a prime and $q \equiv 1 \pmod{2n}$, the multiplication between two ring elements can be performed using a specialized discrete Fourier transform algorithm called the Number Theoretic Transform (NTT).
For v a vector in $\mathbb{Z}_q^n$ and ω a primitive n-th root of unity in $\mathbb{Z}_q^n$ we define $\tilde{v} = \mathtt{NTT}(v)$ the vector such that $\tilde{v}[k] = \sum_{i=0}^{n-1} v[i] \cdot \omega^{ik}$ and $v_{inv} = \mathtt{INTT(v)}$ the vector such that $v_{inv}[k] = n^{-1} \cdot \sum_{i=0}^{n-1} v[i] \cdot \omega^{-ik}$. We have that $v = \mathtt{INTT}(\mathtt{NTT}(v))$ and that for two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_q^n$, $\mathbf{c} = \mathbf{a} * \mathbf{b} = \mathtt{INTT}(\mathtt{NTT}(a) \circ \mathtt{NTT}(b))$ (with $\circ$ denoting the coefficient-wise product) is the cyclic convolution product of a and b.

Interestingly, if we see polynomials as their coefficients vectors in $\mathbb{Z}_q^n$, $a * b$ correspond to the usual multiplication $a \cdot b$ in $\mathbb{Z}_q[X]/\langle X^n - 1\rangle$. To multiply in $\mathcal{R}$ instead of this NTRU-like ring, we have to take care of the minus signs appearing when reducing modulo $X^n + 1$. In order to do that, we must perform a coefficient-wise multiplication between the inputs of the NTT and a vector containing powers of a $2n$-th primitive root of unity ψ and a coefficient-wise multiplication between the output of INTT and a vector containing the inverse powers of ψ.

Putting all together, for ψ a primitive $2n$-th root of unity in $\mathbb{Z}_q$, ψ_{pow} the vector $(1, \psi, \psi^2, ..., \psi^{n-1})$ and ψ_{pow}^{-1} the vector $(1, \psi^{-1}, \psi^{-2}, ..., \psi^{-(n-1)})$, the product of two polynomials $a, b \in \mathcal{R}$ seen as elements in $\mathbb{Z}_q^n$ is given by $a \cdot b = \psi_{pow}^{-1} \circ \mathtt{INTT}(\mathtt{NTT}(a \circ \psi_{pow}) \circ \mathtt{NTT}(b \circ \psi_{pow}))$.
Fortunately, it is possible to directly incorporate the multiplication with ψ_{pow} and ψ_{pow}^{-1} into the computation of the NTT and INTT. Hence, in the following, when referring to those two functions, we means their versions enabling the computation of the product in $\mathcal{R}$. The purpose of using the Number Theoretic Transform to perform products is efficiency, indeed, with FFT algorithms, we can compute NTT and INTT (hence multiplication) in $\mathcal{O}(n \cdot \log n)$. Borrowing terminology from signal processing, we call an output of NTT a vector in frequency domain and an output

of `INTT` (or an untransformed vector) a vector in time domain. In RLWE-based or MLWE-based scheme, a meticulous usage of the time and frequency domain can greatly enhance the speed of the primitive.

2.6 Key encoding and decoding

In LWE-based key exchange protocols, the two parties eventually compute two noisy versions of the exchanged value. Since symmetric algorithms are only useful if the exact same key is used on both side, it required to encode the key in a way allowing correct reconciliation. The usual approach in encryption-based KEM is to encode the key K as a polynomial p with coefficients in $\{0, \lfloor \frac{q}{2} \rfloor\}$. Since those two values are at maximal distance in $\mathbb{Z}_q$, even if an error polynomial e is added to p, it is still possible to retrieve K as long as $\|e\|_\infty \leq \lfloor \frac{q}{4} \rfloor$ using a threshold decoder.

- $p = \texttt{Encode}(K = b_1 b_2 ... b_\lambda) := \sum_i^\lambda b_i \cdot \lfloor \frac{q}{2} \rfloor \cdot X^i$
- $K = \texttt{Decode}(p' = p + e) := \big(b_i = 1 \textbf{ if } \frac{q}{4} < p'_i <= \frac{3q}{4} \textbf{ else } b_i = 0\big)$

2.7 AVR 8-bit microcontrollers

In this work, we target the popular ATmega328p and ATmega2560 microcontrollers found, for example, on the user friendly Arduino Board (respectively UNO and Mega 2560). They are both part of the Atmel AVR family of microcontroller and support the same instruction set. They can be programmed in C/C++ or using AVR assembly language. Most operations operates on 8 bits but there exist a native multiplication operation $8 \times 8 \rightarrow 16$ (but no floating point support). Both microcontrollers run at the same speed but provide a different amount of memory.

	ATmega328p	ATmega2560
SRAM (KB)	2	8
Flash Memory (KB)	32	256
Clock Speed (MHZ)	16	16

3 Module-lattice-based KEM over the ring $\mathcal{R}$

3.1 CPA-KEM from MLWE encryption

Our focus in this paper is a module-LWE-based key encapsulation mechanism constructed from the CPA-secure LWE encryption of Lindner and Peikert[18]. It is described in Algorithms 1, 2 and 3. The variables of the scheme are the following:

- k is the main security parameter. The dimension of the LWE problem seen as an unstructured lattice problem is $k \cdot n \times k \cdot n$.
- n is the dimension of the ring $\mathcal{R} = \mathbb{Z}_q[X]/\langle X^n + 1 \rangle$.
- l controls the size λ of the exchanged secret key in terms of multiples of the ring size, $\lambda = n \cdot l$.
- l' controls the number of coefficients used to encrypt one bit of key.

Algorithm 1 KeyGen

Input: Public parameter $\mathbf{A} \in \mathcal{R}^{k\times k}$

Output: Key pair $pk = \mathbf{T} \in \mathcal{R}^{k\times l}, sk = \mathbf{S} \in \mathcal{R}^{k\times l}$

```
1: $(\mathbf{S}, \mathbf{E}) \leftarrow \mathcal{B}_p^{k\times l} \times \mathcal{B}_p^{k\times l}$
2: $\mathbf{T} \leftarrow \mathbf{A} \cdot \mathbf{S} + \mathbf{E}$
3: return $pk = \mathbf{T}, sk = \mathbf{S}$
```

Algorithm 2 CPA.Encaps

Input: Public parameter $\mathbf{A}$, public key $\mathbf{T}$, random symmetric key $K \in \{0,1\}^{n\cdot l}$

Output: Encrypted key $\mathbf{ct} = (\mathbf{U}, \mathbf{V}) \in \mathcal{R}^{k\times l'} \times \mathcal{R}^{l\times l'}$

```
1: $(\mathbf{R}, \mathbf{E}_1, \mathbf{E}_2) \overset{r}{\leftarrow} \mathcal{B}_p^{k\times l} \times \mathcal{B}_p^{l\times l'} \times \mathcal{B}_p^{l\times l'}$
2: $\mathbf{U} \leftarrow \mathbf{A}^T \cdot \mathbf{R} + \mathbf{E}_1 \in \mathcal{R}^{k\times l'}$
3: $\mathbf{V} \leftarrow \mathbf{T}^T \cdot \mathbf{R} + \mathbf{E}_2 + \mathtt{encode}(K) \in \mathcal{R}^{l\times l'}$
4: return $\mathbf{ct} = (\mathbf{U}, \mathbf{V})$
```

Algorithm 3 CPA.Decaps

Input: Encapsulated Key $\mathbf{ct} = (\mathbf{U}, \mathbf{V})$, secret key $\mathbf{S}$

Output: Symmetric Key K

```
1: $K = \mathtt{decode}(\mathbf{V} - \mathbf{S}^T \cdot \mathbf{U})$
2: return $K$
```

3.2 CCA transformation

As in [8,7,10], we extended it to a CCA-secure KEM by using a generic transformation. We use the same approach as [10] by using the $\mathrm{FO}^{\not\perp}$ transform of [14] proved secure by [16]. We made the small modification of comparing hashes of ciphertexts instead of ciphertexts themselves. It allows us to discard the first ciphertext as soon as we can to limit memory usage in practice. The goal is to avoid the unaffordable situation of having two ciphertexts living in memory at the same time. Another possibility, with even lower memory footprint, could be to rewrite a new encapsulation routine subtracting to the previous ciphertext and verifying that it is equal to zero at the end but we believe it might be a bit far-fetched.

Algorithm 4 CCA.Encaps

Input: Public parameter $\mathbf{A}$, public key $\mathbf{T}$

Output: Key K and its encapsulation $\mathbf{ct}$

```
1: $m \overset{r}{\leftarrow} \{0,1\}^\lambda$
2: $(\hat{K}, r) \leftarrow H(\mathbf{T}, m)$
3: $\mathbf{ct} = \mathtt{CPA.Encaps}(\mathbf{A}, \mathbf{T}, m; r)$
4: $K = H(\hat{K}, \mathbf{ct})$
5: return $(\mathbf{ct}, K)$
```

Algorithm 5 CCA.Decaps

Input: Public parameter $\mathbf{A}$, secret key $sk = (\mathbf{S}, z)$, public key $\mathbf{T}$ and encapsulated key $\mathbf{ct}$

Output: Key K

```
1: $m' \leftarrow \texttt{CPA.Decaps}(\mathbf{ct}, \mathbf{S})$
2: $(\hat{K}', r') \leftarrow H(\mathbf{T}, m')$
3: $\mathbf{ct}' \leftarrow \texttt{CPA.Encaps}(\mathbf{A}, \mathbf{T}, m'; r')$
4: if $H(\mathbf{ct}') = H(\mathbf{ct})$ then
5:     $K = H(\hat{K}, \mathbf{ct})$
6: else
7:     $K = H(z, \mathbf{ct})$
8: return $K$
```

4 Instantiation over $\mathbb{Z}_{257}[X]/\langle X^{128} + 1\rangle$

In order to get a compact instantiation, we decided that a good strategy was to limit the dimension of the polynomials. The idea is that since the algorithms are mainly made of matrix-vector multiplications and dot products between vectors of ring elements, it is possible to get an efficient implementation while only having a minimal number of coefficients unpacked (see 4.2). Obviously, it would be possible to work with all coefficients packed all the time but the performances would be greatly decreased since the NTT is constantly reading and modifying the polynomial. Our implementation works with only one polynomial unpacked at the time. In [7], the authors justified the choice of $n = 256$ by the fact that they wanted to encapsulate a key of 256 bits, one bit per coefficient. While it perfectly makes sense, we decided to work with the ring $\mathbb{Z}_q[X]/\langle X^{128} + 1\rangle$ for the following reasons:

1. It gives more flexibility in the size of underlying LWE instance since we can use all multiples of 128.
2. It enables the NTT algorithm for the very appealing Fermat modulus 257.
3. Its module-LWE problem is less structured.
4. Unpacked polynomials are smaller.
5. It is still possible to exchange 256 bits secrets without increasing to much the memory footprint.[2]

In this work, we consider two sets of parameters detailed in Table 4. In the name KEM_*a*_*b*_*c*, *a* refers to the dimension of the underlying standard LWE instance, *b* is the value of the modulus and *c* is the dimension of the exchanged secret. The reason for having two sets is that while the security of the KEM itself does not depend on the size of the exchanged key, the final goal is often to use a symmetric algorithm afterward and one worried by Grover's algorithm may want to use a 256-bit key.

[2] The keys will be larger but in our case we store them in the less expensive flash memory of the microcontroller

	n	q	k	l	l'	λ	$\lvert\mathbf{ct}\rvert$ (Bytes)
KEM_384_257_128	128	257	3	1	1	128	576
KEM_384_257_256	128	257	3	2	1	256	720

With the first set, the two matrix multiplications are actually a matrix-vector multiplication and a dot product. We can then rewrite the scheme as in Figure 4. We also explicitly wrote the NTT in order to get a clear view of what has to be computed. In the following, the analysis will be made with respect to this set for the sake of brevity but is essentially the same for the second one.

```
KeyGen($\tilde{\mathbf{A}}$) :
  1: $\tilde{\mathbf{s}} \leftarrow$ NTT(Binomial())
  2: $\tilde{\mathbf{e}} \leftarrow$ NTT(Binomial())
  3: $\tilde{\mathbf{t}} \leftarrow \tilde{\mathbf{A}} \cdot \tilde{\mathbf{s}} + \tilde{\mathbf{e}}$
  4: return $\tilde{\mathbf{t}}, \tilde{\mathbf{s}}$
```

```
Encaps($\tilde{\mathbf{A}}, \tilde{\mathbf{t}}, K$) :
  1: $\tilde{\mathbf{r}} \leftarrow$ NTT(Binomial())
  2: $\tilde{\mathbf{e}}_1 \leftarrow$ NTT(Binomial())
  3: $\tilde{e}_2 \leftarrow$ NTT(Binomial())
  4: $\tilde{\mathbf{u}} \leftarrow \tilde{\mathbf{A}}^T \cdot \tilde{\mathbf{r}} + \tilde{\mathbf{e}}_1$
  5: $\tilde{v} \leftarrow \tilde{\mathbf{t}}^T \cdot \tilde{\mathbf{r}} + \tilde{e}_2 +$ Encode($K$)
  6: return $\tilde{\mathbf{u}}, \tilde{v}$
```

```
Decaps($\tilde{\mathbf{s}}, \tilde{\mathbf{u}}, \tilde{v}, K$) :
  1: $K =$ Decode(INTT($\tilde{v} - \tilde{\mathbf{s}} \cdot \tilde{\mathbf{u}}$))
  2: return $K$
```

Fig. 1. Practical instantiation for the set KEM_384_257_128

4.1 Computing modulo 257

When doing arithmetic in $\mathbb{Z}_q[X]/\langle X^n+1\rangle$, it is really convenient to have access to the Number Theoretic Transform to multiply polynomials. Not only it offers a general $\mathcal{O}(n \log n)$ algorithm to multiply ring elements but it also give some optimization opportunities by carefully choosing when to go in and out of the frequency domain. Unfortunately since the transform require primitive $2n$-th root of unity to exist in $\mathbb{Z}_q$, not all values of q are allowed for a fixed ring, the requirement is that $q-1$ divides $2n$. For the ring of dimension 256, the smallest prime having this property is 7681 and is the one used in Kyber. When switching to the ring of dimension 128, we can now set q to the Fermat prime 257. Not only this reduces the size of the polynomials which are now really close to one byte per coefficient but it also enjoy really efficient modular reductions. The trick is to realize that working modulo 257, a $(k+1)$-bit integer $v \equiv \sum_{i=0}^{k} c_i \cdot 2^i \equiv \sum_{i=0}^{7} c_i \cdot 2^i + 2^8 \sum_{j=0}^{k-8} c_j \cdot 2^j \equiv \sum_{i=0}^{7} c_i \cdot 2^i - \sum_{j=0}^{k-8} c_j \cdot 2^j$, using the congruence $2^8 \equiv -1 \mod 257$. It means that the modular reduction of can be computed via `v = (v&0xFF) - (v >> 8)`.
The main operation we need to implement to compute the KEM is the modular multiplication $\mathbb{Z}_{257} \times \mathbb{Z}_{257} \rightarrow \mathbb{Z}_{257}$. The ATmega328p being a 8-bit microcontroller, it is quite cumbersome to work with big integers and we want to limit the size of the intermediate computations to two registers (16 bits). Elements of $\mathbb{Z}_q$ are often naturally represented as integers in the set $\{0, 1, ..., q-1\}$. In our case, the multiplication of two such integers for $q = 257$ would cause an overflow on 16 bits for

$256*256$ which would be inconvenient to handle. Fortunately, arithmetic in $\mathbb{Z}_q$ does not depend on the representative used for the cosets. Hence, we decided to represent $\mathbb{Z}_{257}$ with the signed integers in $S = \{-128, ..., 0, .., 128\}$, this does not impair performances since the microcontroller can natively handle signed multiplications and the multiplication does not overflow on 16 bits anymore. Even better, we can even use representatives in a set slightly larger than S without overflowing, this is really useful to save reductions when adding polynomials from the error distribution to uniformly random ones because we can omit (lazy reduction) to reduce coefficients with absolute values a bit over 128 since they will still be correctly reduced by the multiplication algorithm. The reduction of a 16-bit integer into S is somewhat more involved than just shifting and subtracting but is still really lightweight and requires neither division nor multiplication. The C-like procedure is described in algorithm 6 but a specialized AVR assembly version is used in the code.

Algorithm 6 Reduction of an `int16` to $\{-128, ..., 128\}$ modulo 257

```
function MOD257($v$)
    $v \leftarrow ((v \& 0xFF) - (v >> 8))$
    $v \leftarrow v - (0x101 \& ((0x80 - v) >> 15))$
```

4.2 Representation of packed polynomials

We use two data structures to store polynomials, `Poly = int16[128]` (using 2048 bits) and `Packed_poly = uint8[144]` (using 1152 bits). The goal of `Poly` is to provide an "easy to work with" representation of polynomials of degree 128 over $\mathbb{Z}_{257}$, each coefficient is stored as a 16-bit signed integer in the array. This representation is very loose, a lot of space is wasted since outside of intermediate computations, only 257 different values are needed but accessing a given coefficient is really fast. On the other hand `Packed_poly` is a more compact representation, it is not optimal since the entropy of the polynomial is $\log_2(257) * 128 = 1024.72$ bits but reaching optimality would require a really inefficient procedure to pack and unpack values. Instead we made a reasonable trade-off between ease of encoding and compactness. The 128 first `uint8` represent the 8 least significant bits of the two's complement representation of each coefficient in $[-128, 128]$ and the last 16 values contain sign bits, packed by 8. Hence, even though we use 9 bits to represent 257 values, the unpack procedure only has to retrieve the sign bit with some boolean operations and fill the upper register with copies of it the reconstruct the 16-bit integer.

4.3 Memory efficient algorithm

We now describe how to compute algorithm of Figure 4 with a limited stack usage. Note that the goal here is not to aggressively save every possible byte of RAM but to make a reasonable trade-off permitting to work in memory constraint environments. Pseudo code is given in algorithm 7, let us described the previously undefined functions:

– FILL_ERROR_POLY(p, r): Fill the polynomial p with coefficients sampled from the error distribution using the randomness r

– POINTWISE_ADD_MUL_PACKED(p_1,p_2,p_3): Compute $p_3 = p_1 \cdot p_2 + p_3$ coefficient wise with p_2 and p_3 in packed form.
– ADD_KEY(p, K): Add encoding of the key K to the polynomial p
– XOF(*out*,*outlen*,*in*,*inlen*): Expand a seed *in* of size *inlen* into a pseudo random value *out* of size *outlen*.

Outside temporary values, the only variables living in RAM during the execution are the key K as an array of bytes, x of type `Poly`, $\mathbf{V}$ of type `Packed_poly` and $\mathbf{U}$ as an array of k `Packed_poly`. The public parameter and the public key are both stored in the frequency domain in flash memory. The first part of the algorithm preprocess $\mathbf{V}$ and $\mathbf{U}$ to fill their entries with coefficients from the error distribution (and encodes the key). The purpose of setting the errors before computing the products is to avoid requiring the memory area containing $\mathbf{U}$ and $\mathbf{V}$ to be all zeros since further operations will only add values and never set them. The next part aims to compute both the matrix-vector multiplication $\mathbf{A}^T \cdot \mathbf{r}$ and the dot product $\mathbf{T}^T \cdot \mathbf{r}$ at the same time by iteratively sampling $\mathbf{r}$ entry by entry. At each iteration of the outer loop, one entry of $\mathbf{r}$ is sampled, multiplied by a full row of $\mathbf{A}$ (the matrix is transposed) and by an entry of $\mathbf{T}$, and added respectively to $\mathbf{U}$ and $\mathbf{V}$. Since all polynomials live in the frequency domain, all coefficients are accessed only once during additions and multiplications. For this reason, we can directly apply those operations on packed polynomials by unpacking and repacking each coefficient on the fly without unpacking the whole polynomial.

Algorithm 7 Memory efficient KEM computation

```
XOF(random_tape, random_size, seed, seed_size)
FILL_ERROR_POLY($\mathbf{x}$, random_tape)                        ▷ sample $e_2$
ADD_KEY($\mathbf{x}$, K)                                          ▷ $e_2 + \texttt{Encode}(K)$
NTT($\mathbf{x}$)
PACK($\mathbf{x}$, $\mathbf{V}$[i])
for i=0; i < k; ++i do
    FILL_ERROR_POLY($\mathbf{x}$, random_tape)                    ▷ sample $\mathbf{e}_1$
    NTT($\mathbf{x}$)
    PACK($\mathbf{x}$, $\mathbf{U}$[i])

for i=0; i < k; ++i do
    FILL_ERROR_POLY($\mathbf{x}$, random_tape)                    ▷ sample $\mathbf{r}$
    NTT($\mathbf{x}$)
    for j=0; j < k; ++j do
        POINTWISE_ADD_MUL_PACKED($\mathbf{A}$[i*K + j], $\mathbf{x}$, $\mathbf{U}$[j])   ▷ $\mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1$
    POINTWISE_ADD_MUL_PACKED($\mathbf{T}$[i], $\mathbf{x}$, $\mathbf{V}$)   ▷ $\mathbf{T}^T \cdot \mathbf{r} + e_2 + \texttt{Encode}(K)$
```

4.4 Randomness generation

High quality randomness is always important for cryptographic applications. On UNIX-based systems, it is a common practice to read bytes from `/dev/urandom` to get random numbers. On microcontrollers, the situation is more complex and unfortunately, no algorithmic solution can solve the lack of entropy pool. Since we needed to deterministically regenerate a random tape for the CCA-secure version, we only used `random()` to generate the key and a seed. The seed is deterministically expanded using Salsa20 to generate a pseudo-random stream of bytes. The choice of Salsa20 is somewhat arbitrary and does not affect the core of the implementation (cSHAKE would be another good option) but we decided to reuse the code of the μNaCL library [15] optimized for AVR 8-bit architecture.

4.5 Security

Estimating the exact security of LWE instances is a complex task and is a constantly evolving field. Some estimations make pessimistic analyses based on possible future advances in cryptanalyses while others stick closer to the performances of current algorithms. Fortunately Albrecht and al. [1] developed a tool facilitating the task of assessing bit security of LWE-based schemes. Even though we use a structured problem (module lattices), cryptanalysis algorithm do not take advantage of this fact and the MLWE problem is treated as standard LWE.
We run the estimator with the two following command:

```
n = 384; q = 257; stddev = sqrt(1/2); alpha = alphaf(sigmaf(stddev), q)
_ = estimate_lwe(n, alpha, q, reduction_cost_model=BKZ.sieve)
```

estimating a security of 134 bits and

```
Q_Core_Sieve=lambda beta, d, B: ZZ(2)**RR(0.265*beta)
n = 384; q = 257; stddev = sqrt(1/2); alpha = alphaf(sigmaf(stddev), q)
_ = estimate_lwe(n, alpha, q, reduction_cost_model=Q_Core_Sieve)
```

indicating a quantum security of at least 95 bits using really pessimistic estimations. This places the security above level 1 of NIST.

4.6 Correctness

During decapsulation, the recipient first computes

$$\begin{aligned}\mathbf{v} - \mathbf{s}^T \cdot \mathbf{u} &= \mathbf{t} \cdot \mathbf{r} - \mathbf{A} \cdot \mathbf{s} \cdot \mathbf{r} - \mathbf{s}^T \cdot \mathbf{e}_1 + e_2 + \texttt{Encode}(K) \\ &= \mathbf{A} \cdot \mathbf{s} \cdot \mathbf{r} + \mathbf{e} \cdot \mathbf{r} - \mathbf{A} \cdot \mathbf{s} \cdot \mathbf{r} - \mathbf{s}^T \cdot \mathbf{e}_1 + e_2 + \texttt{Encode}(K) \\ &= \mathbf{e} \cdot \mathbf{r} - \mathbf{s}^T \cdot \mathbf{e}_1 + e_2 + \texttt{Encode}(k)\end{aligned}$$

then uses the `Decode()` algorithm and hopefully recovers the correct key. Since it is a simple threshold decoder without any additional error-correcting procedure, the keys exchanged are the same if all coefficients of $\Delta = \left|\mathbf{e} \cdot \mathbf{r} - \mathbf{s}^T \cdot \mathbf{e}_1 + \mathbf{e}_2\right|$ are less than $\lfloor \frac{q}{4} \rfloor$. Let us estimate the correctness of the scheme by computing the probability of $\|\Delta\|_\infty < 64$. Since this polynomial is the result of the sum of two dot products and an error polynomial, each of its coefficient is a sum of $128 \cdot 2 \cdot 3 = 768$ random variables from the product of $\mathcal{B}_1$ with itself (let us call it $\mathcal{B}_1^2$) and a random variable from $\mathcal{B}_1$. Estimating bounds on the sum of random variable is not always easy but in case of small supports, it is possible to explicitly compute all convolutions. The distribution $\mathcal{B}_1^2$ is easy to determine by hand. Let $A \sim \mathcal{B}_1$ and $B \sim \mathcal{B}_1$:

$$\mathbb{P}\left[AB = 0\right] = \frac{3}{4}$$

$$\mathbb{P}\left[AB = z \mid z \in \{-1, 1\}\right] = \frac{1}{8}$$

We wrote a Python script using Numpy to compute the 768 convolutions by $\mathcal{B}_1^2$ in a square-and-multiply fashion and convoluted a last time with $\mathcal{B}_1$ to take into account e_2. We then applied the union bound on the 128 coefficients of Δ to determine the overall failure probability. We determined that the scheme is $2^{-10.7}$-correct. This is clearly a limitation in comparison to general purpose KEMs.

5 Experimental results

Experimental results are presented in Table 1. The first set of parameters has been benchmarked on the ATmega328p and the second one on the ATmega2560. We can see that the core of the key encapsulation, that is to say the module-lattice-based CPA encryption, offers satisfying performances, even in such a constraint environment. Regarding memory footprint, we were glad to be able to fit the algorithm in under 2KB of RAM, making its computation on the small ATmega328P possible. The unsatisfactory result is the runtime of the CCA version. Since it uses CPA.Encaps and CPA.Decaps as black boxes, its inefficiency is due to the computation of the hash functions on a whole ciphertext. We used the well known BLAKE32 and SHA3 since our work in totally orthogonal to this choice but a lightweight alternative might be to consider.[3]

Parameters	Algorithm	Speed (ms)	Total stack (Bytes)
KEM_384_257_128	CPA.Encaps	64.36	1320
	CPA.Decaps	12.44	944
	CCA.Encaps	106.70	1519
	CCA.Decaps	194.87	1613
KEM_384_257_256	CPA.Encaps	77.98	1497
	CPA.Decaps	20.36	1161
	CCA.Encaps	269.52	2022
	CCA.Decaps	638.94	2016

Table 1. Experimental results on ATmega328p and ATmega2560.

6 Conclusion

In this work, we studied a key encapsulation mechanism derived from Kyber and Frodo with parameters tailored for embedded systems and more specifically small 8-bit popular AVR microcontrollers. As a novelty, we decided to work over a ring of dimension 128. This allows us to reduce the number of unpacked coefficients concurrently living in memory and also to use the NTT compatible modulus 257. This Fermat prime enables a really efficient reduction procedure which is welcome since the AVR architecture does not support neither vectorized instructions nor floating point operations. We assessed the security of those parameters using common

[3] See [5] for an extensive comparison of hash functions on AVR.

estimation techniques and corroborate their legitimacy by providing experimental results on ATmega328p and ATmega2560. The price to pay is a reduced correctness of the scheme which prevents it to be used in all contexts. Nevertheless, it provides an interesting trade-off for the IoT world.

References

1. M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. `https://bitbucket.org/malb/lwe-estimator`.
2. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Newhope without reconciliation. Cryptology ePrint Archive, Report 2016/1157, 2016. `http://eprint.iacr.org/2016/1157`.
3. E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe. Post-quantum key exchange—a new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, 2016. USENIX Association.
4. E. Alkim, P. Jakubeit, and P. Schwabe. Newhope on arm cortex-m. In C. Carlet, M. A. Hasan, and V. Saraswat, editors, *Security, Privacy, and Applied Cryptography Engineering*, pages 332–349, Cham, 2016. Springer International Publishing.
5. J. Balasch, B. Ege, T. Eisenbarth, B. Gérard, Z. Gong, T. Güneysu, S. Heyse, S. Kerckhof, F. Koeune, T. Plos, T. Pöppelmann, F. Regazzoni, F.-X. Standaert, G. Van Assche, R. Van Keer, L. van Oldeneel tot Oldenzeel, and I. von Maurich. Compact implementation and performance evaluation of hash functions in attiny devices. In S. Mangard, editor, *Smart Card Research and Advanced Applications*, pages 158–172, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
6. A. Boorghany, S. B. Sarmadi, and R. Jalili. On constrained implementation of lattice-based cryptographic primitives and schemes on smart cards. *ACM Trans. Embed. Comput. Syst.*, 14(3):42:1–42:25, Apr. 2015.
7. J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, and D. Stehlé. Crystals – kyber: a cca-secure module-lattice-based kem. Cryptology ePrint Archive, Report 2017/634, 2017. `http://eprint.iacr.org/2017/634`.
8. J. W. Bos, C. Costello, L. Ducas, I. Mironov, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo : Take off the ring ! practical , quantum-secure key exchange from lwe. 2016.
9. J. Buchmann, F. Göpfert, T. Güneysu, T. Oder, and T. Pöppelmann. High-performance and lightweight lattice-based public-key encryption. In *Proceedings of the 2Nd ACM International Workshop on IoT Privacy, Trust, and Security*, IoTPTS '16, pages 2–9, New York, NY, USA, 2016. ACM.
10. J.-P. D'Anvers, A. Karmakar, S. Sinha Roy, and F. Vercauteren. Saber: Module-lwr based key exchange, cpa-secure encryption and cca-secure kem. In A. Joux, A. Nitaj, and T. Rachidi, editors, *Progress in Cryptology – AFRICACRYPT 2018*, pages 282–305, Cham, 2018. Springer International Publishing.
11. R. de Clercq, S. S. Roy, F. Vercauteren, and I. Verbauwhede. Efficient software implementation of ring-lwe encryption. DATE '15, pages 339–344, San Jose, CA, USA, 2015. EDA Consortium.
12. E. Fujisaki and T. Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 537–554, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
13. T. Güneysu and T. Oder. Towards lightweight identity-based encryption for the post-quantum-secure internet of things. *2017 18th International Symposium on Quality Electronic Design (ISQED)*, pages 319–324, 2017.
14. D. Hofheinz, K. Hvelmanns, and E. Kiltz. A modular analysis of the fujisaki-okamoto transformation. Cryptology ePrint Archive, Report 2017/604, 2017. `https://eprint.iacr.org/2017/604`.

15. M. Hutter and P. Schwabe. NaCl on 8-bit AVR microcontrollers. In A. Youssef and A. Nitaj, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 156–172. Springer-Verlag Berlin Heidelberg, 2013. Document ID: cd4aad485407c33ece17e509622eb554, `http://cryptojedi.org/papers/#avrnacl`.
16. H. Jiang, Z. Zhang, L. Chen, H. Wang, and Z. Ma. Ind-cca-secure key encapsulation mechanism in the quantum random oracle model, revisited. Cryptology ePrint Archive, Report 2017/1096, 2017. `https://eprint.iacr.org/2017/1096`.
17. A. Langlois and D. Stehle. Worst-case to average-case reductions for module lattices. Cryptology ePrint Archive, Report 2012/090, 2012. `https://eprint.iacr.org/2012/090`.
18. R. Lindner and C. Peikert. Better key sizes (and attacks) for lwe-based encryption. In A. Kiayias, editor, *Topics in Cryptology – CT-RSA 2011*, pages 319–339, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
19. Z. Liu, T. Pöppelmann, T. Oder, H. Seo, S. S. Roy, T. Güneysu, J. Großschädl, H. Kim, and I. Verbauwhede. High-performance ideal lattice-based cryptography on 8-bit avr microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 16(4):117:1–117:24, July 2017.
20. Z. Liu, H. Seo, S. Sinha Roy, J. Großschädl, H. Kim, and I. Verbauwhede. Efficient ring-lwe encryption on 8-bit avr processors. In T. Güneysu and H. Handschuh, editors, *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 663–682, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
21. V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, Nov. 2013.
22. T. Oder, T. Pöppelmann, and T. Güneysu. Beyond ecdsa and rsa: Lattice-based digital signatures on constrained devices. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 110:1–110:6, New York, NY, USA, 2014. ACM.
23. T. Pöppelmann, T. Oder, and T. Güneysu. High-performance ideal lattice-based cryptography on 8-bit atxmega microcontrollers. In K. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology – LATINCRYPT 2015*, pages 346–365, Cham, 2015. Springer International Publishing.
24. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93, New York, NY, USA, 2005. ACM.
25. P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, Oct. 1997.