Master Thesis
Samuele Andreoli

Introduction

Contents

Chapter 1

Theoretical Background

Chapter 2

Previous PQKE

2.1 The BCNS proposal

Chapter 3

New Hope and Frodo

3.1 New Hope

In the New Hope paper[ADPS16], the authors present some solutions to performance and security issues of the BCNS proposal [BCNS14] of an Unauthenticated Post Quantum Key Exchange suitable for use in the TLS cipher suite, deriving the New Hope protocol.
As far as performance are concerned, the main improvements are a change in the error distribution, from Gaussian, to a much easier to sample Binomial, the use of a new lattice for error reconciliation, that allows for better error correction, and a drastic drop in the size of the modulus q, and finally the use of an encoding for polynomials in the NTT[1] domain, which allows to take advantage of the quasi linear product in the NTT domain, while skipping some of the NTT transforms that would usually be necessary.
As for security, the authors notice that fixing the parameter $\mathbf{a}$ creates an opportunity for an all-for-the-price-of-one attack, and advice to make it ephemeral in order to avoid such risk.
Moreover, the authors present a comprehensive security analysis, assessing the security level of the protocol to be comfortably over 128 bit.
Let's take the time to go through the proposed changes and examine each of them in detail.

3.1.1 Error Distribution

Previous proposals were considerably burdened by the choice of a Gaussian noise distribution, which is notoriously hard to sample. The authors of New Hope opt for a much easier to sample Binomial distribution Ψ_k of parameter $k = 16$. Such distribution is really easy to sample from a stream of uniform i.i.d. random bits, which is commonly available on modern machines.
The security of the key exchange is only marginally affected by the choice of this distribution, as we can see in Theorem 1, for the proof of which we refer to [ADPS16].

Theorem 1. *Let ξ be the rounded Gaussian distribution of parameter $\sigma = \sqrt{8}$, let $\mathscr{P}$ be the idealized version of protocol 3.1, using the distribution ξ instead of*

[1] TODO Reference to the section with the NTT

$\Psi_1 6$. *If an unbounded algorithm succeeds in recovering the pre-hash key ν, given a transcript of an instance of protocol 3.1, then it would also succeed against $\mathscr{P}$ with probability $q \geq p^{9/8}/26$.*

3.1.2 Recovery from errors

In R-LWE applications, usually, one bit of information is encoded in each of the polynomial coefficients. In the context of a key exchange, though, this translates to a message space larger than necessary. This is an opportunity to introduce redundancy in the transmitted message (i.e. the key), trying to increase error tolerance. The authors of New Hope provide a generalization of a bit encoding using two coefficients, presented in [TODO add citation], which allows to encode a bit in 4 coefficients, further increasing the error tolerance w.r.t to the previous method using two coefficients. It is worth pointing out that the encoding part is not really important as far as the New Hope protocol is concerned. Indeed, the polynomials are always created either as result of operations on other polynomials, or via sampling from some distribution. Decoding, on the other hand, is used to extract the bits of the pre-hash key from the polynomials **v** and **v'**, and is thus a crucial step of the reconciliation procedure. In order to build this encoding, a polynomial $f(X) \in \mathscr{R} = \mathbb{Z}[X]/(X^n + 1)$ has to be split in $n/4$ groups of 4 coefficients. Then, bits need to be encoded into, and decoded from, groups of four coefficients.

Polynomial splitting

The first goal, i.e. the polynomial splitting, can be quite naturally accomplished observing that, given $\mathscr{S} = \mathbb{Z}[X]/(X^4 + 1)$, we can write $f(X) \in \mathscr{R}$ as $f(X) = f_0 + f_1 X + \ldots + f_{1023} X^{1023} = f'_0(X^{256}) + \ldots + f'_{255}(X^{256})X^{255}$, where the f'_i are elements of $\mathscr{S}$, thus identifying $f(X)$ with the vector $(f'_0, \ldots, f'_{255}) \in \mathscr{S}^{256}$. This means that each f'_i is the polynomial of coefficients $f_i, f_{i+256}, f_{i+512}, f_{i+768}$, which can be identified with a vector in $\mathbb{Z}^4$.

Bit encoding

For the encoding, the authors propose to use the lattice $\tilde{D}_4 = \mathbb{Z}^4 \cup \mathbf{g} + \mathbb{Z}^4$, where $\mathbf{g}$ is the glue vector $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2})^t$, such lattice has Voronoi cell $\mathscr{V}$ with relevant vectors $(\pm 1, 0, 0, 0), (0, \pm 1, 0, 0), (0, 0, \pm 1, 0), (0, 0, 0, \pm 1)$, said type-A, and $(\pm\frac{1}{2}, \pm\frac{1}{2}, \pm\frac{1}{2}, \pm\frac{1}{2})$, said type-B, respectively 8 and 16 in number.
The condition for a correct error recovery, given the cell $\mathscr{V}$, is that given the error vector e, then $e \in \mathscr{V}$. This condition can be rewritten as $\langle e, v \rangle \leq \frac{1}{2} \forall v$ relevant vectors. The distinction between the type-A and type-B can be used to cleverly regroup these equations, obtaining $\|e\|_\infty \leq \frac{1}{2}$ for type-A vectors, and $\|e\|_1 \leq 1$ for type-B vectors.
The last component for the encoding is a base for $\tilde{D}_4$. The authors of [ADPS16] propose the base $\mathbf{B} = (\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{g})$. Although this might not seem the most natural choice, the presence of the glue vector gives an efficient algorithm for decoding over $\tilde{D}_4/\mathbb{Z}^4$.
Indeed, considering that $\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, 2\mathbf{g} \in \mathbb{Z}^4$, a vector in $\tilde{D}_4/\mathbb{Z}^4$ is given by the parity of its last coordinate. This means that the encoding of a bit $k \in \mathbb{Z}_2$ can be computed as in Algorithm 3.1.3, as $k\mathbf{g}$. The decoding is described in

Algorithm 3.1.4, and it can be summed up as looking for the closest vector to the element $\mathbf{x} \in \tilde{D}_4$ to decode, between $\mathbf{0}$ and $\mathbf{g}$. It is easy to see that if $e \in \mathscr{V}$, then $\mathbf{x} + e$ will still be in the same Voronoi cell of $\mathbf{x}$.

Reconciliation

Finally, all of the ingredients for the reconciliation mechanism are in place. The closest vector any $\mathbf{x} \in \mathbb{R}$ can be computed using the $CVP_{\tilde{D}_4}$ algorithm, descirbed in Algorithm 3.1.2, and with this the r-bit reconciliation function `HelpRec` can be defined as

$$CVP_{\tilde{D}_4}\left(\frac{2^r}{q}(\mathbf{x} + b\mathbf{g})\right) mod 2^r, where \xleftarrow{\$} U[\{0,1\}] \tag{3.1}$$

Let's call $\mathbf{r}$ the output of the reconciliation function. The final extracted bits can then be computed as

$$\texttt{Decode}\left(\frac{1}{q}\mathbf{x} - \mathbf{Br}\right) \tag{3.2}$$

Applying this algorithm to the polynomial split computed above, the full key can be recovered.
The vector r, output of the `HelpRec` function, will then be transmitted to the other party, allowing it to apply the final reconciliation to its polynomial split, thus recovering the same key.

Failure rate

3.1.3 Generation of a

The parameter $\mathbf{a}$ was made ephemeral, in order to avoid all-for-the-price-of-one attacks and parameter manipulation[2]. Generating a fresh $\mathbf{a}$ for each run of the protocol, opens two problems. The generation of $\mathbf{a}$ is computationally somewhat costly, and the generated parameter has to be transmitted to the other party of the key exchange, posing a bandwidth problem.
The former issue can be mitigated by caching the parameter $\mathbf{a}$ for multiple instances of the protocol, since it has only a marginal effect on the resilience against all-for-the-price-of-one attack. As we can see in Protocol 3.1, this is quite useful for Alice, which is identified as the Server in a Client-Server connection, and could still be of some use to Bob, which is the Client.
The latter is solved by generating the parameter $\mathbf{a}$ from a small random seed, using an extendable output cryptographic hash function, SHAKE-128[3], to generate a sequence of pseudo-random bits, which are then parsed into the coefficients of the polynomial $\mathbf{a}$, as described in Algorithm 3.1.1. The generated $\mathbf{a}$ is considered to be in the NTT domain, so there is not need to apply the transform to it. This is a legitimate choice, because the NTT transform preserves uniform noise, so no bias is introduced in the coefficients of $\mathbf{a}$ by considering it already in the NTT domain.
It's worth to point out that although `Parse` rejects some of the material generated by SHAKE-128, the rejection rate is only $(2^{16} - 1 - 5q)/(2^{16} - 1) \approx 0.06$.

[2]For example, the parameter $\mathbf{a}$ can be trapdoored, as described in [GPV07] and [ADPS16].
[3]This is an extension of the SHA-3 cryptographic hash function [SHA15]

Since **a** has $n = 1024$ coefficients, and each of them is a 16 bit unsigned integer, SHAKE-128 needs to generate ≈ 2.2 kbytes of output on average.

Algorithm 3.1.1. *Parse function*

Data: S, a bit sequence, the output of SHAKE-128
Result: **a**, a polynomial, represented as an array of its coefficients
$i \leftarrow 0$
$c \leftarrow 0$
while $i \leq n$ **do**
  $Xi \leftarrow (uint16)S[16 * c..16 * c + 15]$
  $c++$
  if $Xi \leq 5 * q$ **then**
    $\mathbf{a}[i] \leftarrow Xi$
    $i++$
  end if
end while

Algorithm 3.1.2. $CVP_{\tilde{D}_4}$

Data: $\mathbf{x} \in \mathbf{R}^4$
Result: $\mathbf{z} \in \mathbb{Z}^4$ *s.t.* $\mathbf{x} - \mathbf{Bz} \in \mathscr{V}$
$\mathbf{v}_0 \leftarrow \lfloor \mathbf{x} \rceil$
$\mathbf{v}_1 \leftarrow \lfloor \mathbf{x} - \mathbf{g} \rceil$
if $\|\mathbf{x} - \mathbf{v}_0\|_1 < 1$ **then**
  $k \leftarrow 0$
else
  $k \leftarrow 1$
end if
$(\nu_0, \nu_1, \nu_2, \nu_3)^t \leftarrow \mathbf{v}_k$
return $(\nu_0, \nu_1, \nu_2, k)^t + \nu_3 \cdot (-1, -1, -1, 2)^t$

Algorithm 3.1.3. *Encode*

Data: $k \in \{0, 1\}$
Result: $\mathbf{x} \in \mathbf{R}^4/\mathbb{Z}^4$, the encoding
return $k \cdot \mathbf{g}$

NTT and message encoding

As mentioned above,

3.1.4 Protocol run

In Protocol 3.1 we can see the end result of the proposed improvements: the New Hope protocol for a 128 bit security level, with public parameters $p = 12289$, the modulus, and $n = 1024$, the degree of the polynomials.

Algorithm 3.1.4. *Decode*

Data: $\mathbf{x} \in \mathbf{R}^4/\mathbb{Z}^4$
Result: $k \in \{0,1\}$ *s.t.* $\mathbf{x} - k\mathbf{g} \in \mathscr{V} + \mathbb{Z}^4$
  $\mathbf{v} \leftarrow \mathbf{x} - \lfloor \mathbf{x} \rceil$
  if $\|\mathbf{v}\|_1 \leq 1$ **then**
    return 0
  else
    return 1
  end if

Protocol 3.1. *New Hope*

Alice	Bob
$seed \xleftarrow{\$} U[\{0,1\}^{256}]$	
$\mathbf{a} \leftarrow \texttt{Parse}(\texttt{SHAKE-128}(seed)))$	
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \psi_{16}^n$	$\mathbf{s'}, \mathbf{e'}, \mathbf{e''} \xleftarrow{\$} \psi_{16}^n$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	
$(seed, \mathbf{b}) \rightarrow$	
	$\mathbf{a} \leftarrow \texttt{Parse}(\texttt{SHAKE-128}(seed)))$
	$\mathbf{u} \leftarrow \mathbf{as'} + \mathbf{e'}$
	$\mathbf{v} \leftarrow \mathbf{bs'} + \mathbf{e''}$
	$\mathbf{r} \xleftarrow{\$} \texttt{HelpRec}(\mathbf{v})$
	$\leftarrow (\mathbf{u}, \mathbf{r})$
$\mathbf{v'} \leftarrow \mathbf{us}$	
$\nu \leftarrow \texttt{Rec}(\mathbf{v'}, \mathbf{r})$	$\nu \leftarrow \texttt{Rec}(\mathbf{v}, \mathbf{r})$
$\mu \leftarrow \texttt{SHA3-256}(\nu)$	$\mu \leftarrow \texttt{SHA3-256}(\nu)$

3.1.5 Security analysis

3.2 Going back to LWE

3.3 Frodo

The authors of Frodo [BCD+16] restart from the BCNS proposal [BCNS14], using a more general LWE setting, instead of the R-LWE used in the original paper. Then, as in the New Hope paper, they propose improvements to the protocol, to make it viable for use in the TLS cipher suite.
Since the same security issues pointed out in [ADPS16] are still valid in the LWE setting, the authors of Frodo are forced to make the parameter $\mathbf{a}$ ephemeral, which has an even bigger cost than in the R-LWE setting, both in terms of bandwidth and computational cost. As in New Hope, the first issue is solved using a small seed to generate the parameter $\mathbf{a}$, but the size of this parameter in the setting of LWE can still be an issue for more restricted environments, so the authors of Frodo decided to use a generation function that allows for generation of small blocks of $\mathbf{a}$, which can be used and then discarded if caching them in memory is too burdensome. Even after the optimization, the generation process is remarkably costly, taking up to 40% of the time necessary. Following the same reasoning of Section 3.1.3, caching might be a partial mitigation for this burden.
Moreover, the authors present four different easy to sample noise distributions

which might be used in place of the rounded Gaussian distribution used in [BCNS14].

3.3.1 Reconciliation

The reconciliation mechanism for Frodo is a generalization of the mechanism presented in [Pei14]. This generalized mechanism allows packing one or more bit in each approximate agreed element of $\mathbb{Z}_q$[4], similarly to the reconciliation mechanism of New Hope. On the other hand, while in New Hope there was plenty of room to add some redundancy in the agreed polynomial, this is a luxury the authors of Frodo couldn't afford. On the contrary, for realistic security levels more bits need to be packed in each approximate agreed value. In fact, in order to achieve a security level of s bits, we need to choose $\bar{n}$, $\bar{m}$ and B, the number of bits packed into an agreed value, such that $s \geq \bar{m}\bar{n}B$. Increasing B allows to reduce $\bar{m}$ and $\bar{n}$, i.e. the dimension of the LWE matrices, thus decreasing not only the computational cost, but also the bandwidth requirements of the algorithm.
Consider the quantity B and $\bar{B} = log_2[q] - B$. Two functions are defined starting from these two quantities: the *rounding* function, as

$$\begin{aligned} \lfloor \cdot \rceil_{2^B} : \mathbb{Z}_q &\to [0, 2^B - 1] \\ v &\mapsto \lfloor 2^{-\bar{B}} v \rceil \bmod 2^B \end{aligned}$$

and the *cross-rounding* function, as

$$\begin{aligned} \langle \cdot \rangle_{2^B} : \mathbb{Z}_q &\to \{0, 1\} \\ v &\mapsto \lfloor 2^{-\bar{B}+1} v \rfloor \bmod 2 \end{aligned}$$

Using these two functions, the `rec` function can be defined as in [Pei14]

$$\begin{aligned} \mathrm{Rec} : \mathbb{Z}_q \times 0, 1 &\to [0, 2^B - 1] \\ (w, b) &\mapsto \lfloor v \rceil_{2^B} \end{aligned}$$

where v is the closest vector to w s.t. $\langle v \rangle_{2^B} = b$.

3.3.2 Error distributions

As mentioned above, the authors present four different easy to sample probability distribution that could be used instead of the rounded Gaussian. In order to achieve a reasonable degree of efficiency both in terms of computational cost of sampling, and of entropy required to generate the samples, the authors decide to use inverse sampling, using a pre-computed table for a cumulative density function, CDF from now on, carefully chosen to be as close as possible to a rounded Gaussian distribution[5]. We refer to [BCD+16] for the exact tables proposed by the authors of Frodo.
Let's examine the procedure for inverse sampling used in Frodo. For instance, take the first distribution[6] proposed in [BCD+16]. This distribution has CDF

[4] In this paragraph, q is chosen to be a power of two, but this mechanism can be adapted to work with any modulus.
[5] The authors use the notion of Rènyi divergence to quantify closeness. More details in Section 3.3.5
[6] Called D_1 in the original paper.

given by $T = [43, 104, 124, 127]$. In order to inverse sample, generate an uniformly random value $y \in [0, 127]$, i.e. seven random bits, and then return the smallest $\bar{x} \in [0, 3]$ *s.t.* $y \leq T[\bar{x}]$. Such $\bar{x}$ can only be positive, while the rounded Gaussian also takes negative values. So, in order to "restore" the sign, an eighth uniformly random bit is generated and used to choose $s \in \{-1, 1\}$, thus generating the final value $x = s\bar{x}$.
In general, the construction of a table can be viewed as taking 2^{b-1} samples from a rounded Gaussian[7] and counting the occurrences of the different absolute values. The derived PDF is then converted into a CDF. Since the sign is ignored, the CDF does not resemble a rounded Gaussian anymore, but it is easy to restore the sign, with just an additional random bit, while this procedure allows for a more compact CDF, with only half the values. This procedure has a total requirement of b bits of entropy, and it only costs a constant time table lookup, which runs in $\Theta(\#T)$, and the sign multiplication.

3.3.3 Generation of a

As mentioned above, a fresh matrix $\mathbf{a} \in \mathbb{Z}_q^{n \times n}$ needs to be generated for each protocol instance. The approach followed in Frodo is similar to the one of New Hope. A small uniformly random seed is generated and shared between the parties[8], who can generate the same $\mathbf{a}$ using it. Following the same reasoning of [ADPS16], the seed is expanded using a function that fits the random oracle model. In particular, the function used for the task is AES128 in ECB mode, using the seed as key and a known plaintext.
The procedure starts from a "striped" matrix, $\bar{\mathbf{a}}$, *s.t.*

$$\bar{\mathbf{a}}[i, j] = \begin{cases} i & if\, j = 0 \bmod 8 \\ j - 1 & if\, j = 1 \bmod 8 \\ 0 & otherwise \end{cases}$$

which translates in a matrix with the following shape.

$$\bar{\mathbf{a}} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 8 & 0 & \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 8 & 0 & \cdots \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 8 & 0 & \\ & & & & & \vdots & & & & & & \\ n-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & n-1 & 8 & 0 & \cdots \end{bmatrix}$$

In $\bar{\mathbf{a}}$, the elements are represented as 16 bit unsigned integers, memorized in little-endian order[9]. Then, the elements of a row are taken in groups of 8 and passed into AES128 in ECB mode, using the seed as key for the cipher, deriving the final $\mathbf{a}$. Notice that the arrays used as plaintext are of the form $[i, 8 \cdot k, 0, \ldots, 0]$ *s.t.* $i \in \{0, \ldots, n\}, k \in \{0, \ldots, n/8\}$, which are all unique, since a pair (i, k) is never repeated. Because of this, and of the random oracle heuristic, the authors justify using this matrix in place of a randomly sampled matrix

[7] This is just for reference, the CDF are usually hand crafted
[8] Remember that the parameter $\mathbf{a}$ is public
[9] This allows for a natural implementation on little-endian machines

with uniform distribution.
In Algorithm 3.3.1, there is the pseudocode to generate the whole matrix $\mathbf{a}$. It is easy to see that generating the whole matrix is quite burdensome, as this procedure need to instantiate at least one $n \times n$ matrix[10], or two, if it skips the instantiation of $\bar{\mathbf{a}}$ by keeping a pre-generated copy of it. This procedure is particularly unfeasible for restricted environments, where the memory is a precious resource. For instance, using the parameters recommended in [BCD+16], such matrix uses a total amount of $752 * 752 * 16bit \approx 1.1MB$ of memory.
All is not lost. This construction of the matrix $\mathbf{a}$, indeed, allows for individual blocks of the matrix to be individually generated quite efficiently, to be then used and discarded when they are not needed anymore. There are two particular ways of generating blocks of the matrix, which are extremely useful for our purpose, i.e. multiplicating the matrix $\mathbf{a}$ with other matrices[11]. Indeed, the matrix can be generated by rows, as we can see in Algorithm 3.3.2 and by columns, as we can see in Algorithm 3.3.3. The generated row, or columns, can be used in the matrix multiplication and then safely discarded, thus slashing the memory consumption for the generation of $\mathbf{a}$, reducing it by a factor of $n/8 = 94$ for the generation by columns, to $\approx 12KB$, and by a factor of $n = 752$, to $\approx 1.5KB$, for the generation by rows[12].
It is worth pointing out that in the context of TLS the Server might want to cache the parameter $\mathbf{a}$ to use it for multiple connections over a set time share, as suggested at the beginning of 3.3. Since Servers usually have a decent amount of memory, they can afford to store the full generated matrix, thus saving about 40% of the computational cost for a protocol instance[13]. On the other hand, Clients have more limited resources, and almost no benefit from caching, so the generation on the fly is generally preferable for them, both in terms of computational cost and memory usage.

3.3.4 Protocol run

In protocol 3.2 we can see a run of Frodo, for a security level s, with public parameters $(n, q, \chi, \bar{n}, \bar{m}, B)$. For simplicity, the matrix $\mathbf{a}$ is generated with `Gen` as described in Algorithm 3.3.1 and subsequently multiplied to create the public terms for the key exchange. It is then easy to adapt the protocol to generate blocks of $\mathbf{a}$ on the fly when executing the matrix multiplication.

3.3.5 Security Analysis

[10] Indeed, the matrix $\mathbf{a}$ can overwrite $\bar{\mathbf{a}}$, so a second instantiation is not necessary
[11] Both on the left and on the right. These matrices have lower dimensions, thus being easier to store in memory
[12] Assuming the recommended parameters in [BCD+16] are used
[13] Using a pre-generated $\mathbf{a}$. The cost of generating $\mathbf{a}$ on the go is actually lower that generating the full matrix.

Algorithm 3.3.1. *Generate a*

Data: $seed \in \{0,1\}^{128}$
Result: **a**, an $n \times n$ matrix generated from the seed
/* Instantiate $\bar{\mathbf{a}}$ */
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    if $j = 0 \bmod 8$ **then**
      $\bar{\mathbf{a}}[i,j] \leftarrow i$
    else if $j = 1 \bmod 8$ **then**
      $\bar{\mathbf{a}}[i,j] \leftarrow j-1$
    else
      $\bar{\mathbf{a}}[i,j] \leftarrow 0$
    end if
  end for
end for
/* Generate **a** */
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-8$ **step** 8 **do**
    $\mathbf{a}[i, j:j+8] \leftarrow AES128(\bar{\mathbf{a}}[i, j:j+8], seed)$
  end for
end for
return **a**

Algorithm 3.3.2. *Generate* $\boldsymbol{a}$ *by rows*

Data: $seed \in \{0,1\}^{128}, i \in \{0,\ldots,n-1\}$, the row index
Result: **a**[i,:], an array of n elements generated from the seed
for $j \leftarrow 0$ **to** $n-8$ **step** 8 **do**
  $\mathbf{a}_i[j:j+8] \leftarrow AES128([i,j,0,0,0,0,0,0], seed)$
end for
return $\mathbf{a}_i$

Algorithm 3.3.3. *Generate* $\boldsymbol{a}$ *by columns*

Data: $seed \in \{0,1\}^{128}, j \in \{0,8,\ldots,n-8\}$, the starting column index
Result: **a**[:,j:j+8], an $n \times 8$ matrix generated from the seed
for $i \leftarrow 0$ **to** $n-1$ **do**
  $\mathbf{a}_j[i, j:j+8] \leftarrow AES128([i,j,0,0,0,0,0,0], seed)$
end for
return $\mathbf{a}_j$

Protocol 3.2. *Frodo*

Alice [Server]	Bob [Client]
$seed \xleftarrow{\$} U[\{0,1\}^s]$	
$\mathbf{a} \leftarrow \mathtt{Gen}(seed)$	
$\mathbf{s}, \mathbf{e} \xleftarrow{\$} \chi[\mathbb{Z}_q^{n \times \bar{n}}]$	$\mathbf{s'}, \mathbf{e'} \xleftarrow{\$} \chi[\mathbb{Z}^{\bar{m} \times n_q}]$
	$\mathbf{e''} \xleftarrow{\$} \chi[\mathbb{Z}_q^{\bar{m} \times \bar{n}}]$
$\mathbf{b} \leftarrow \mathbf{as} + \mathbf{e}$	
$(seed, \mathbf{b}) \rightarrow$	$\mathbf{a} \leftarrow \mathtt{Gen}(seed)$
	$\mathbf{u} \leftarrow \mathbf{s'a} + \mathbf{e'}$
	$\mathbf{v} \leftarrow \mathbf{s'b} + \mathbf{e''}$
	$\mathbf{r} \xleftarrow{\$} \langle \mathbf{v} \rangle_{2^B}$
$\mathbf{v'} \leftarrow \mathbf{us}$	$\leftarrow (\mathbf{u}, \mathbf{r})$
$\nu \leftarrow \mathtt{Rec}(\mathbf{v'}, \mathbf{r})$	$\nu \leftarrow \lfloor \mathbf{v} \rceil_{2^B}$

Chapter 4

Implementation

4.1 Computational Complexity

The computational complexity of Frodo is dominated by the parameter generation, with a cost of $O(n^2)$, and the public key generation, with a cost of $O(\bar{m}n^2)$ or $O(\bar{n}n^2)$. Luckily, we tipically have $\bar{m}, \bar{n} \ll n$, which means that the total cost of the algorithm cn be summed up as $O(n^2)$. This cost does not look promising, but these parameters are tipically very small, with n being the highest, but still remaining abundantly lower that 1000. Because of these small parameters, the actual cost of the protocol is remarkably small. Indeed, Frodo is just slightly slower than New Hope, and orders of magnitude faster than the competitor PQKE.
The real hit to the performance of Frodo comes from the memory usage and the bandwith. Again, the culprits are the parameter generation and the matrix multiplications.
As far as memory is concerned, the naive implemenation is still dominated by the memory necessary for the parameter a, which is $O(n^2)$, and the memor necessary for the private and public keys, which is $O(n)$[1]. In Section 3.3.3 there is an in-depth analysis of the cost, and the possible mitigations for the generation of the parameter a, which practically reduces the memory requirements to $O(n)$.
Bandwidth is another weak spot of Frodo, as the matrices that need to be transmitted are not small either. As seen above, these grow as $O(n)$, which translates in a pretty hefty amount of data to be exchanged.

4.2 Implementation Rationale

As the main goal of this thesis is not to examine the differences of Frodo and New Hope only from a theoretical point of view, but also to give a practical comparison of the two, an original implementation is presented. It focuses on portability, low memory usage, both in terms of executable size and execution memory and It uses stack memory in all its internal functions, naturally avoiding issues related to memory handling, so frequent in a language like C.
The nature of the LWE primitives naturally suits a constant time implementation for the key generation and reconciliation primitives, and with a little more

[1] Considering, again, that $\bar{m}, \bar{n} \ll n$

effort all the critical parts of the protocol can be made completely branchless, with only a few loops, which only depend on public parameters and not on execution data, neither private nor public.
With this in mind, the proposed implementation focuses on streamlining the one proposed by the authors of [BCD+16], pruning unnecessary branches and options that are a burden during execution and make the code less readable.
Let's examine in detail the different aspects of the protocol implementation.

4.2.1 Element representation

The core choice we need to make when implementing Frodo is element representation, which can either be tight or redundant. A tight representation would save memory, but it would burden the implementation with the modular arithmetic handling. On the other hand, a redundant representation would ease this burden, either by using lazy reduction, or by carefully selecting modulo and bits so that overflows can be exploited to automatically take care of modular arithmetic. This comes at the cost of an increased memory usage.
The authors of [BCD+16] have no doubts on their choice, and their intents are quite clear from the parameters choice. They don't even consider using lazy reduction and force the modulus to be a power of two, thus enabling the implementation to ignore the modular arithmetic. Indeed, considering that the highest possible modulus in any parameter choice is 2^{15}, an element of a matrix can always be represented as an integer in $\mathbb{Z}_{16}$. This means that an element can always fit an unsigned 16 bit integer, regardless of the parameter choice, and that the modular arithmetic is taken car of by integer overflow.
The final advantage of this representation comes when an element needs to be reduced to fit in $\mathbb{Z}_n$: a single AND instruction with the mask $2^n - 1$ is enough to compute the desired value. This final computation might not even be necessary, as it is often the case that either specific bits are extracted from the 16 bits integer itself, or that the integers are packed into byte arrays for transmission. In these cases, it is possible to perform the bitwise operations in such a way that the modular arithmetic is automatically handled.

4.2.2 Parameter generation

The parameter generation, alongside key pairs generation, is the most burdensome part of Frodo. Although, not much can be added to the considerations made in Section 3.3.3 about the generation of the parameter, either in full, or by row or column. It is only worth mentioning that while key generation needs to be handled with care to mitigate side channel attacks, the generation of the public parameter is not so critical. There is no advantage in loosening the requirements on constant timeness, but there are small gains from avoiding some of the final cleanups of the memory used in some parts of the procedure,. For instance, the buffers used in the AES encryption of the parameter stripes can be safely left alone.

4.2.3 Matrix multiplication

The shares recombination and the public key computation are nothing more than matrix multiplication, with the addition of the sampled noise matrices. Unfortunately, the dimension of the matrices used in Frodo is not high enough to justify more advanced methods of matrix multiplication, and the naive algorithm is the preferred choice. Still, the approach followed by the authors of Frodo for theses procedures can be improved. Let's examine how the original implementation behaves.
The tipical operation that needs to be executed when muliplying two matrices in Frodo, is not just a simple multiplication, but it also requires the addition of a sampled noise matrix, i.e. $ab+e$. In the original implementation, all the three matrices a, b, and e are allocated, plus the destintion matrix. Then, the matrix e is copied into the destination matrix, and the product is computed and stored directly in the destination matrix.
It is easy to see that one of these allocations is unnecessary: the matrix e is allocated, used to store the samples and then copied over to the destination. So, why not making the matrix e itself the destination matrix. This approach saves the allocation of a matrix, which is quite burdensome, and also the time necessary to copy over the matrix e to the destination. Since the error matrices are generated on the fly and never reused, nothing is lost by destroying e, and it can save up to one third of the necessary memory, in the case of the multiplication with the parameter generated by rows. Even if the product does not require the addition of noise, there is no extra cost in zeroing the matrix before passing it to the multiplication utility, w.r.t zeroing out the matrix as we go during the product.

4.2.4 Reconciliation

The implementation proposed by the authors of [BCD+16] for the reconciliation follows the definition quite closely. First, the elements of the recombined share are crossrounded as described in Section **??**, with or without the help of the hint, depending on which party is executing the reconciliation. Then, the desired bits are extracted from the matrices and packed into the agreed key, using generic packing utilities[2].

4.2.5 Element packing

Although a redundant representation may be a good choice for internal element representation, it is not acceptable when the data has to be transmitted to the other party in the key exchange. This issue would add to the already hefty memory and bandwidth cost of Frodo. In order to address this problem, the elements in redundant representation are packed into the tight representation before transmission, and unpacked when received.
This part of the protocol is not really expensive, but there is much to improve in the original implementation. Indeed, the authors of [BCD+16] try to fit the code

[2] More on these utilities in the next section

for bit packing both to the element packing and the key packing from the reconciled combined shares. This overburdens the implementation with unnecessary complexity, forcing a packing algorithm that operates bit by bit, and ultimately resulting in fairly inefficient packing and unpacking procedures. Moreover, this choice might raise few eyebrows because it is used both for packing the public shares and the private reconciled secret. Decoupling the logic that handles private and public material allows for a simpler, branchless, implementation for the former, as seen in Section 4.2.4, and a more efficient implementation for the latter, scratching the requirements for constant timeness and memory safety.
To address this issue, this implementation proposes a bytewise packing procedure. Unfortunately, the parameter choice does not allow for the most efficient branchless implementation, except for the Frodo Classic parameters, but

Conclusions

Bibliography

[ADPS16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange—a new hope. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 327–343, Austin, TX, 2016. USENIX Association.

[BCD+16] Joppe Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In *23rd ACM Conference on Computer and Communications Security (ACM CCS)*, pages 1006–1018, 2016.

[BCNS14] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-quantum key exchange for the tls protocol from the ring learning with errors problem. Cryptology ePrint Archive, Report 2014/599, 2014.

[GPV07] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. 2007.

[Pei14] Chris Peikert. Lattice cryptography for the internet. In *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, pages 197–219, 2014.

[SHA15] Sha-3 standard: Permutation-based hash and extendable-output functions. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. FIPS PUB 202, 2015.