kyber

July 9, 2022

1 Simplified Kyber - Beispiel mit Implementierung in Python

2 1.1 Was ist Kyber?

- Gewinner der NIST-Ausschreibung für PQC-Verfahren (Post-quantum cryptography)
  - Begründung für den Sieg: "comparatively small encryption keys [...] its speed of operation."
- Ist Teil der "Cryptographic Suite for Algebraic Lattices" (CRYSTALS), CRYSTALS-Dilithium (PQ-Signatur-Verfahren), ebenfalls Gewinner in der NIST-Ausschreibung
- Public Key Encryption System -> Key-Paare
- KEM -> Key encapsulation mechanism (Verschlüsselung von symmetrischen Schlüsseln)
  - Genaue Spezifikation heißt "IND-CCA2-secure key-encapsulation mechanism"
  - "Indistinguishability under adaptive chosen ciphertext attack"
- Sicherheit von Kyber basiert auf dem MLWE-Problem (Module learning with errors)
- Praktischer Einsatz möglich trotz längerer Schlüssel, immernoch weniger als andere PQC-Finalisten
  - Vgl. Classic McEliece -> Key size: > 1MB
- Die Operationen auf dem Polynomring (Multiplikation von großen Integern und Polynomen mit hohem Grad) lassen sich mit Hilfe von NTT (Number theoretic transforms) sehr stark optimieren

2.1 1.1.1 Schlüssellängen

Version	Sicherheitslevel	Private Key	Public Key	Ciphertext	NIST Sicherheitslevel
Kyber512	AES128	1632	800	768	1
Kyber768	AES192	2400	1184	1088	3
Kyber1024	AES256	3168	1568	1568	5
RSA3072	AES128	384	384	384	1
RSA7680	AES192	960	960	960	3
RSA15360	AES256	1920	1920	1920	5

*Größenangaben sind in Byte

3 1.2 Mathematische Vorraussetzungen

- Alle Berechnungen finden in dem Polynomring $R = \mathbb{Z}[X]/(X^n + 1)$ und $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, wobei $n = 256$ und $q = 7681$ ist.
- Jedes Polynom $p \in R_q$ kann wie folgt dargestellt werden: $p = (\sum_{i=0}^{n-1} a_i * x^i) mod q$

4 1.3 Module learning with errors

Einfach zu lösen ist das Problem: Gegeben sei $A \in R_q^{kxk}$ mit zufällig gewählten Polynomen und $a_{i,j} \in R_q^{kx1}$ mit $i, j \in \{0, k-1\}$, dann lässt sich $\begin{pmatrix} a_{1,1} & \dots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{k,1} & \dots & a_{k,k} \end{pmatrix} \begin{pmatrix} s_1 \\ \vdots \\ s_k \end{pmatrix} = \begin{pmatrix} t_1 \\ \vdots \\ t_k \end{pmatrix}$ der Vektor s mit dem gaußschen Eliminationsverfahren in $O(n^3)$ berechnen. Das Problem wird unverhältnismäßig schwierig, wenn man noch einen Error-vektor e mit "kleinen" Koeffizienten dazunimmt. Sei nun $e \in R_q^{kx1}$ mit zufällig gewählten "kleinen" Polynomen, dann nimmt man an, dass man t mit $\begin{pmatrix} a_{1,1} & \dots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{k,1} & \dots & a_{k,k} \end{pmatrix} \begin{pmatrix} s_1 \\ \vdots \\ s_k \end{pmatrix} + \begin{pmatrix} e_1 \\ \vdots \\ e_k \end{pmatrix} = \begin{pmatrix} t_1 \\ \vdots \\ t_k \end{pmatrix}$ nicht mehr in polynomieller Laufzeit bestimmen kann. Über einen nicht trivialen Beweis lässt sich dieses Problem in das SVP (Shortest vector problem) überführen, von dem man weiß, dass dieses selbst für Quantencomputer NP-hard ist.

5 1.4 Unterschiede zum regulärem Kyber:

- Kleinere Parameter für bessere Lesbarkeit
- Kompression des Chiffretexts - beinflusst nicht das zugrundeliegende Kryptosystem
- Verwendung der Zufallsgeneratoren von Python anstelle vom in Kyber vorgestellten Sampling
- Keine Verwendung von NTT für Berechnung von Polynomen

6 2.1 Key Generation

6.1 2.1.1 Modul q, mit q ist eine Primzahl

Warum ist $q = 3329$?

- Initial wurde q mit 7681 angesetzt um die Bedigung $q \ mod \ 2n = 1$ zu erfüllen (Ermöglicht Optimierungen bei der Multiplikation)
- In der 2. Runde wechselte man zu 3329, da man herausfand, dass auch $q \ mod \ n = 1$ reicht um gleiche, bzw. bessere Performance zu erreichen

Es wird $q = 17$ gesetzt und weil viel mit Polynomen gerechnet, Darstellung als Polynom: $f = x^4 + 1$. Für die Multiplikation von Zahlen wird q und für die Multiplikation mit Polynomen wird f verwendet.

[1]:
```
from sympy import Poly
from sympy.abc import x, y
from sympy import GF
```

```
from sympy import QQ
import sympy as S
import numpy as np
import math
import decimal

q = 17 # Modul
dom = GF(q, symmetric=False) # Galoisfeld als endlicher Zahlenkoerper
f = Poly(1 + x**4, domain=dom) # Binäre Representation von Q als Polynom
zero = Poly(0, x, domain=dom)
print('q: ', q)
print('f: ', f)
print(dom)

# Im Original-Paper wird bei einem "tie" aufgerundet
# z.B. 8.5 wird zu 9 aufgerundet
decimal.getcontext().rounding = decimal.ROUND_HALF_UP
def round(number):
    return int(decimal.Decimal(number).to_integral_value())
```

```
q:  17
f:  Poly(x**4 + 1, x, modulus=17)
GF(17)
```

6.2 2.1.2 Private key s

Sei $s = \begin{pmatrix} -x^3 - x^2 + x \\ -x^3 - x \end{pmatrix}$

[2]:
```
s = np.array([[Poly(-x**3 - x**2 + x)], [Poly(-x**3 - x)]])
print('s: ', s, '\ns-dimension: ', s.shape) # Spaltenvektor
```

```
s:  [[Poly(-x**3 - x**2 + x, x, domain='ZZ')]
 [Poly(-x**3 - x, x, domain='ZZ')]]
s-dimension:  (2, 1)
```

6.3 2.1.3 Public key (A, t)

Sei $A = \begin{pmatrix} 6x^3 + 16x^2 + 16x + 11 & 9x^3 + 4x^2 + 6x + 3 \\ 5x^3 + 3x^2 + 10x + 1 & 6x^3 + x^2 + 9x + 15 \end{pmatrix}$ $e = \begin{pmatrix} x^2 \\ x^2 - x \end{pmatrix}$ $t = As + e = \begin{pmatrix} 16x^3 + 15x^2 + 7 \\ 10x^3 + 12x^2 + 11x + 6 \end{pmatrix}$

[3]:
```
A = np.array([[Poly(6*x**3 + 16*x**2 + 16*x + 11, domain=dom), Poly(9*x**3 +␣
↪4*x**2 + 6*x + 3, domain=dom)],
              [Poly(5*x**3 + 3*x**2 + 10*x + 1, domain=dom), Poly(6*x**3 +␣
↪1*x**2 + 9*x + 15, domain=dom)]])
print('A: ', A, '\nA-dimension: ', A.shape)
```

```
e = np.array([[Poly(x**2)], [Poly(x**2 - x)]])
print('e: ', e, '\ne-dimension: ', e.shape)
```

```
A:  [[Poly(6*x**3 + 16*x**2 + 16*x + 11, x, modulus=17)
  Poly(9*x**3 + 4*x**2 + 6*x + 3, x, modulus=17)]
 [Poly(5*x**3 + 3*x**2 + 10*x + 1, x, modulus=17)
  Poly(6*x**3 + x**2 + 9*x + 15, x, modulus=17)]]
A-dimension:  (2, 2)
e:  [[Poly(x**2, x, domain='ZZ')]
 [Poly(x**2 - x, x, domain='ZZ')]]
e-dimension:  (2, 1)
```

[8]:
```
def poly_mul(x, y):
    rows = x.shape[0]
    cols = y.shape[1]
    y_rows = y.shape[0]
    result = np.full((rows, cols), zero) # Array mit 0 initialisieren
    for i in range(rows):
        for j in range(cols):
            for k in range(y_rows):
                result[i][j] = result[i][j].add((x[i][k].mul(y[k][j]))) #␣
 ↪Skalarprodukt
            result[i][j] = result[i][j].rem(f) # Modulo Polynom F
            result[i][j] = Poly(result[i][j], domain=dom) # Modulo Primzahl Q
    return result

def poly_add(x, y):
    rows = x.shape[0]
    cols = y.shape[1]
    result = np.empty((rows, cols), Poly)
    for i in range(rows):
        for j in range(cols):
            result[i][j] = Poly(x[i][j].add(y[i][j]), domain=dom) # Modulo␣
 ↪Primzahl Q
    return result

def poly_sub(x, y):
    rows = x.shape[0]
    cols = y.shape[1]
    result = np.empty((rows, cols), Poly)
    for i in range(rows):
        for j in range(cols):
            result[i][j] = Poly(x[i][j].sub(y[i][j]), domain=dom) # Modulo␣
 ↪Primzahl Q
    return result

t = poly_mul(A, s)
```

```
t = poly_add(t, e)

pk = (A,t)
sk = s

print('A: ', A, '\nA-dimension: ', A.shape)
print('t: ', t, '\nt-dimension: ', t.shape)
print('s: ', s, '\ns-dimension: ', s.shape)
```

```
A:  [[Poly(6*x**3 + 16*x**2 + 16*x + 11, x, modulus=17)
  Poly(9*x**3 + 4*x**2 + 6*x + 3, x, modulus=17)]
 [Poly(5*x**3 + 3*x**2 + 10*x + 1, x, modulus=17)
  Poly(6*x**3 + x**2 + 9*x + 15, x, modulus=17)]]
A-dimension:  (2, 2)
t:  [[Poly(16*x**3 + 15*x**2 + 7, x, modulus=17)]
 [Poly(10*x**3 + 12*x**2 + 11*x + 6, x, modulus=17)]]
t-dimension:  (2, 1)
s:  [[Poly(-x**3 - x**2 + x, x, domain='ZZ')]
 [Poly(-x**3 - x, x, domain='ZZ')]]
s-dimension:  (2, 1)
```

7 2.2 Encryption

Wie üblich in Kryptosystemen, wird eine Nachricht m mit einem Public Key verschlüsselt. Zusätzlich zu dem PK-Tupel (A, t) werden auch noch ein Error- und Zufalls-Polynom-Vektoren e_1 und r benötigt, welche für jede Verschlüsselung zufällig neu generiert werden. Außerdem braucht man noch ein Error-Polynom e_2.

$$r = \begin{pmatrix} -x^3 + x^2 \\ x^3 + x^2 - 1 \end{pmatrix} \; e_1 = \begin{pmatrix} x^2 + x \\ x^2 \end{pmatrix} \; e_2 = -x^3 - x^2$$

[4]:
```
r = np.array([[Poly(-x**3 + x**2)], [Poly(x**3 + x**2 - 1)]])
e_one = np.array([[Poly(x**2 + x)], [Poly(x**2)]])
e_two = Poly(-x**3 - x**2)
print('r: ', r, '\nr-dimension: ', r.shape)
print('e1: ', e_one, '\ne1-dimension: ', e_one.shape)
print('e2: ', e_two)
```

```
r:  [[Poly(-x**3 + x**2, x, domain='ZZ')]
 [Poly(x**3 + x**2 - 1, x, domain='ZZ')]]
r-dimension:  (2, 1)
e1:  [[Poly(x**2 + x, x, domain='ZZ')]
 [Poly(x**2, x, domain='ZZ')]]
e1-dimension:  (2, 1)
e2:  Poly(-x**3 - x**2, x, domain='ZZ')
```

Um eine Nachricht m zu verschlüsseln bringt man diese erst in Binärdarstellung und verwendet die Bits als Koeffizienten. $m = 11$, $(11)_{10} = (1011)_2$ $m_b = 1x^3 + 0x^2 + 1x^2 + 1x^0 = x^3 + x + 1$

[5]:
```
m = 11
mb = np.array([int(x) for x in np.binary_repr(m)])
print(mb)
```

```
[1 0 1 1]
```

Die Nachricht als Binärpolynom muss nun noch um den Faktor $\lfloor\frac{q}{2}\rceil$ hochskaliert werden. $m_{bs} = \lfloor\frac{q}{2}\rceil * m_b = 9 * m_b = 9x^3 + 9x + 9$ Warum man die Nachricht skaliert, wird bei der Entschlüsselung ersichtlich.

[6]:
```
mbs = round(q/2) * mb
mbs_poly = Poly(mbs, x)
print(mbs_poly)
```

```
Poly(9*x**3 + 9*x + 9, x, domain='ZZ')
```

Das Ergebnis der eigentlichen Verschlüsselung, also der Chiffretext c, besteht aus dem Tupel $c = (u, v)$. $u^T = r^T A + e_1^T$ $v = r^T t + e_2 + m_{bs}$

$u = \begin{pmatrix} 11x^3 + 11x^2 + 10x + 3 \\ 4x^3 + 4x^2 + 13x + 11 \end{pmatrix}$ $v = 8x^3 + 6x^2 + 9x + 16$

[9]:
```
u = poly_add(poly_mul(r.transpose(), A), e_one.transpose()).transpose()
print('u: ', u, '\nu-dimension: ', u.shape)
# [0][0], da das Polynom in einem 1x1 Vektor steht und man das tatsächlich␣
↪Polynom zur weiteren Berechnung benötigt
v = np.array([[Poly(poly_mul(r.transpose(), t)[0][0] + e_two + mbs_poly,␣
↪domain=dom)]])
print('v: ', v)
```

```
u:  [[Poly(11*x**3 + 11*x**2 + 10*x + 3, x, modulus=17)]
 [Poly(4*x**3 + 4*x**2 + 13*x + 11, x, modulus=17)]]
u-dimension:  (2, 1)
v:  [[Poly(8*x**3 + 6*x**2 + 9*x + 16, x, modulus=17)]]
```

8 2.3 Decryption

8.1 2.3.1 Nachricht mit Störsignal

Die Entschlüsselung kann nun nur die Persion durchführen, welche den SK s kennt. Um die verschlüsselte Nachricht $c = (u, v)$ nun zu entschlüsseln muss man folgendes berechnen: $m_n = v - u * s \Leftrightarrow m_n = r^T * t + e_2 + m_{bs} - (r^T * A + e_1^T) * s \Leftrightarrow m_n = r^T * (A * s + e) + e_2 + m_{bs} - (r^T * A + e_1^T) * s$ $\Leftrightarrow m_n = r^T * A * s + r^T * e + e_2 + m_{bs} - r^T * A * s - e_1^T * s \Leftrightarrow m_n = r^T * e + e_2 + m_{bs} - e_1^T * s$

Jetzt ist auch vielleicht ersichtlich warum man die Nachricht hochskaliert hat. In m_n sind die Koeffizienten aller Terme ,außer m_{bs}, klein. Also kann man die Nachricht selbst mit dem Störsignal $r^T * e + e_2 - e_1^T * s$ wiederherstellen, indem man für jeden Koeffizienten schaut ob dieser näher an $\lfloor\frac{q}{2}\rceil$ oder an 0, bzw. q ist.

[10]:
```
mn = poly_sub(v, poly_mul(s.transpose(), u))[0][0]
print('mn: ', mn)
```

```
mn:  Poly(8*x**3 + 14*x**2 + 8*x + 6, x, modulus=17)
```

8.2 2.4 Wiederherstellen der Nachricht

$\lfloor \frac{q}{2} \rceil = 9$ $q = 17$ $m_n = 8x^3 + 14x^2 + 8x + 6$

- 8, näher an 9 als an 0/q, nach 9 runden
- 14, näher an q als an 9, nach 0 runden
- 8, näher an 9 als an 0/q, nach 9 runden
- 6, näher an 9 als an 0/q, nach 9 runden

Man erhält also: $m_{bs} = 9x^3 + 9x + 9$ $m_{bs} = \lfloor \frac{q}{2} \rceil * m_b$ $m_b = 1x^3 + 0x^2 + 1x + 1$ Aus m_b kann man nun die Bits der originalen Nachricht wieder ablesen und man erhält m = $(1011)_{10} = (11)_2$.

9 3 Parameter sets

Parameter set	n	k	q	η	(d_u, d_v)	δ
Kyber512	256	2	3329	2	$(10, 3)$	2^{-178}
Kyber768	256	3	3329	2	$(10, 4)$	2^{-164}
Kyber1024	256	4	3329	2	$(10, 5)$	2^{-174}

- n: So gewählt damit man 256 Bits Entropie erreicht, kleiner/größere Werte würden die Sicherheit/Skalierbarkeit beinträchtigen
- k: Dimension der Vektoren/Matrizen
- q: Modul, Begründung steht oben
- η und (d_u, d_v): Beide Parameter skalieren den Kompressionsfaktor
- δ: Wahrscheinlichkeit für das Auftreten einer fehlerhaften Entschlüsselung

10 4 Zusammenfassung

- Kyber ist ein Public Key KEM System
- Sicherheit von Kyber basiert auf dem MLWE Problem
- Baut auf dem LPR (Lyubashevsky, Peikert, Regev) Encryption scheme auf, jedoch leichte Veränderungen
- Lässt sich sehr effizient implementieren

11 5 Quellen

- https://cryptopedia.dev/posts/kyber/
- https://pq-crystals.org/kyber/
- https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf
- https://github.com/VadimLyubash/non-app-KyberSaber/blob/main/non-app.pdf
- https://www.nist.gov/news-events/news/2022/07/nist-announces-first-four-quantum-resistant-cryptographic-algorithms
- https://eprint.iacr.org/2017/634.pdf
- https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf
- https://pq-crystals.org/kyber/data/kyber-specification-round3-20210131.pdf