Capítulo 1

Explorando esquemas criptográficos pós-quânticos considerados pelo NIST com implementação em Sage

Thales Paiva (CASNAV e Fundep), Vitor Ponciano (CASNAV, Fundep e UFRJ), Everaldo Moreira (CASNAV e UNICAMP), Rafael Oliveira (CASNAV e IME), Vilc Rufino (CASNAV), Cabral Melo (UFRJ), Julio López (UNICAMP), Eduardo Ueda (SENAC e IPT) e Routo Terada (USP)

Abstract

This chapter presents four post-quantum cryptography schemes that NIST considers to be promising: NTRU, Crystals-Kyber, Crystals-Dilithium, and HQC. To introduce the mathematical foundation of the schemes, we take a concrete approach by providing SageMath code that allows newcomers to the field to further explore with the mathematical objects. We cover a broad range of topics ranging from Algebra and efficient implementation of polynomial operations with the Number Theory Transform to Reed-Solomon codes and advanced cryptographic constructions.

Resumo

Este capítulo apresenta quatro algoritmos criptográficos pós-quânticos que o NIST considera promissores: NTRU, Crystals-Kyber, Crystals-Dilithium, e HQC. Para apresentar os fundamentos dos esquemas considerados, propomos uma abordagem prática em que descrevemos o código em SageMath no meio da discussão matemática, o que permite que pesquisadores interessados na área possam explorar os conceitos enquanto aprendem. Os esquemas escolhidos permitem uma discussão abrangente de tópicos, indo de Álgebra e a implementação eficiente de operações polinomiais com a Transformada da Teoria dos Números, até códigos de Reed-Solomon e construções criptográficas avançadas.

O presente trabalho foi realizado com o apoio da Financiadora de Estudos e Projetos (FINEP).

©2023 Sociedade Brasileira de Computação

1.1. Introdução

O trabalho seminal de Shor apresentou novos algoritmos quânticos aplicáveis aos problemas da fatoração de inteiros e do logaritmo discreto [Shor 1994]. Embora ainda não sejam conhecidos computadores quânticos grandes e confiáveis o suficiente para aplicar esses algoritmos contra problemas reais, é possível que futuros avanços na construção de tais computadores tornem vulneráveis alguns dos esquemas criptográficos mais usados hoje, como os de curvas elípticas e o RSA.

Esquemas criptográficos para os quais não se conhecem ataques quânticos eficientes são chamados pós-quânticos. Com o objetivo de acelerar a implantação de esquemas pós-quânticos, o National Institute of Standards and Technology (NIST) deu início, em 2016, um processo de padronização de tais esquemas [Chen et al. 2016]. O processo foca em primitivas essenciais para a comunicação segura na Internet, como assinatura digital e troca de chaves. Este processo é dividido em etapas, em que os esquemas melhores avaliados avançam rumo à padronização.

Hoje o processo do NIST está em sua 4ª. rodada [Alagic et al. 2022], e já foram selecionados esquemas para serem padronizados, tanto para troca de chaves, como o Crystals-Kyber [Avanzi et al. 2019], quanto para assinatura digital, como o Cyrstals-Dilithium [Bai et al. 2021], o Falcon [Prest et al. 2020], e o SPHINCS+ [Bernstein et al. 2019b]. O NIST considera padronizar outros esquemas para troca de chaves escolhidos dentre o BIKE [Aragon et al. 2022], o HQC [Melchor et al. 2021], e o Classic McEliece [Bernstein et al. 2019a], chamados alternativos. Note que há esquemas de troca de chaves fora da lista do NIST considerados tão válidos quanto o Crystals-Kyber [Avanzi et al. 2019], como o NTRU [Hoffstein et al. 1998] e o Saber [D'Anvers et al. 2018], porém que foram preteridos por critérios de desempenho.

Objetivo: Apresentar uma seleção de esquemas criptográficos pós-quânticos, considerados promissores pelo NIST, de forma que os participantes os conheçam e possam experimentá-los. Os esquemas selecionados foram: (*i*) NTRU [Chen et al. 2020], (*ii*) Crystals-Kyber [Avanzi et al. 2019], (*iii*) Crystals-Dilithium [Bai et al. 2021], e (*iv*) HQC [Melchor et al. 2021].

Notas sobre o código: O capítulo apresenta os códigos em SageMath dos esquemas escolhidos, contudo essas implementações possuem caráter didático e não devem ser aplicadas diretamente em sistemas em produção. Todo o código mostrado neste capítulo está disponível em `https://github.com/thalespaiva/pqsage`.

1.2. Conceitos fundamentais

1.2.1. Anéis comutativos e corpos finitos

Esquemas de criptografia de chave pública, em geral, fundamentam sua segurança na dificuldade de resolver problemas matemáticos reconhecidamente difíceis. Porém, para que um problema matemático difícil seja útil, é requerido que sua representação computacional seja factível e eficiente.

©2023 Sociedade Brasileira de Computação

Para definir muitos dos problemas matemáticos em que os esquemas atuais são baseados, precisamos trabalhar com conjuntos que admitem operações como soma e multiplicação. Conjuntos em que a multiplicação é distributiva sobre a soma, e onde ambas as operações são comutativas e associativas são chamados anéis comutativos.

Neste texto, há dois tipos de anéis comutativos em que nos interessamos: anéis de inteiros e anéis polinomiais. Dado um inteiro positivo q, denotamos por $\mathbb{Z}_q$ o conjunto $\mathbb{Z}_q = \{0, 1, \ldots, q-1\}$. Se $a, b \in \mathbb{Z}_q$, então as operações $a+b$, $a-b$ ou ab são feitas módulo q. Considere o exemplo em Sage abaixo em que definimos $\mathbb{Z}_{10}$. Aproveitamos para mostrar como ver a tabela de multiplicação, que ilustra como todas as operações são feitas módulo 10.

```
1 sage: Zq = Integers(10)
2 sage: Zq.multiplication_table('digits')
```

Note que, em $\mathbb{Z}_{10}$, alguns elementos têm inversa e outros não. Por exemplo, $3^{-1} = 7$, pois $3 \cdot 7 \equiv 1 \mod 10$. Porém 2 não é inversível pois $2a \not\equiv 1 \mod 10$ para todo $a \in \mathbb{Z}_{10}$. Em geral, se $a \in \mathbb{Z}_q$, então a é inversível se somente se $\gcd(a, q) = 1$.

Dado qualquer anel R, podemos definir um anel polinomial $R[x]$, que consiste em polinômios na variável x de coeficientes em R. Apesar de este conjunto ser infinito, podemos usá-lo para construir um anel finito de forma análoga ao que usamos para os anéis finitos de inteiros. Se tomarmos um polinômio $m(x) \in R[x]$, podemos construir o anel quociente $R[x]/m(x)$ da seguinte forma: $R[x]/m(x) = \{p \bmod m(x) : p(x) \in R[x]\}$. Neste caso, se $a(x), b(x) \in R[x]/m(x)$, então as operações $a(x) + b(x)$ e $a(x)b(x)$ são feitas módulo $m(x)$. Note que, neste texto, muitas vezes omitiremos a variável x, denotando o polinômio $a(x)$ simplesmente por a.

Neste texto, em geral estaremos interessados em anéis da forma $\mathbb{Z}_q[x]/(x^n + 1)$ ou $\mathbb{Z}_q[x]/(x^n - 1)$. Uma vantagem destes anéis é que a redução módulo $(x^n + 1)$ ou $(x^n - 1)$ é fácil pois $x^n \equiv 1 \mod (x^n - 1)$ e $x^n \equiv -1 \mod (x^n + 1)$.

Em Sage, podemos construir o anel $\mathbb{Z}_{11}[x]/(x^4 - 1)$ e realizar experimentos utilizando o código fornecido a seguir.

```
1 sage: PolyRing = PolynomialRing(Integers(11), 'x')
2 ....: x = PolyRing.gen()
3 ....: Rq = PolyRing.quotient(x^4 - 1, 'x')
4 ....: a = Rq.random_element()
5 ....: b = Rq.random_element()
6 ....: print(f'{a = }\n{b = }\n{a * b = }')
7 a = x^3 + 7*x^2 + 7*x + 5
8 b = 3*x^3 + 7*x^2 + 2*x + 7
9 a * b = 8*x^3 + 2*x^2 + 10*x + 8
```

Em $\mathbb{Z}_{11}$, o polinômio $(x^4 - 1)$ pode ser fatorado em $(x+1)(x+10)(x^2+1)$, então existem polinômios em $\mathbb{Z}_{11}[x]/\left(x^4 - 1\right)$ que não têm inversa. Em particular, múltiplos dos fatores de $(x^4 - 1)$ não são inversíveis. Polinômios que não podem ser fatorados em polinômios não constantes com coeficientes em $\mathbb{Z}_q$ são ditos irredutíveis em $\mathbb{Z}_q$.

Anéis em que todos os elementos não-nulos são inversíveis são chamados corpos. Se q é um número primo, então $\mathbb{F}_q = \mathbb{Z}_q$ é um corpo finito de ordem q. O Kyber e o Dilithium usam anéis polinomiais sobre corpos finitos da forma $\mathbb{F}_q[x]/(x^{256} + 1)$ para diferentes valores de q, enquanto o HQC é usa o corpo binário $\mathbb{F}_2$ e usa o anel polinomial

$\mathbb{F}_q[x]/(x^r - 1)$.

O exemplo abaixo mostra como construir o corpo finito $\mathbb{F}_q$ para o primo $q = 13$. Note como de fato $4 \cdot 10 = 40 \equiv 1 \mod 13$.

```
1 sage: F = GF(13)
2 ....: a = F.random_element()
3 ....: print(f'{a = }, {a.inverse() = }')
4 a = 4, a.inverse() = 10
```

Há corpos finitos de ordem não prima $q = p^m$, onde p é primo. Em particular para nosso texto, um dos códigos corretores de erro usados no HQC é definido sobre $\mathbb{F}_{2^8} = \mathbb{F}_2[x]/\left(x^8 + x^4 + x^3 + x^2 + 1\right)$. Este anel polinomial forma um corpo finito pois o polinômio $x^8 + x^4 + x^3 + x^2 + 1$ é irredutível sobre $\mathbb{F}_2$. Isso faz com que, para todo polinômio $a \in \mathbb{F}_2[x]$ de grau menor do que 8, o máximo divisor comum entre a e $x^8 + x^4 + x^3 + x^2 + 1$ seja 1. Como há $2^8 = 256$ polinômios de grau menor que 8 em $\mathbb{F}_2[x]$, este corpo é chamado de $\mathbb{F}_{2^8}$. O corpo finito $\mathbb{F}_{2^8}$ é particularmente importante para computação, pois cada um de seus polinômios pode ser associado a um byte. Intuitivamente, podemos pensar que $\mathbb{F}_{2^8}$ define operações de soma e multiplicação, ambas inversíveis, sobre bytes.

Todos os esquemas de que trataremos neste texto usam alguma noção de tamanho de polinômios ou vetores. Dado um polinômio $a = a_0 + \ldots + a_{n-1}x^{n-1} \in \mathbb{Z}_q/m(x)$, definimos a sua norma ifinito como $\|a\|_\infty = \max\{|\bar{a}_i| : i = 0, \ldots, n-1\}$, onde $\bar{a}_i = a_i \mod q$ e $-\lfloor q/2 \rfloor \leq \bar{a}_i < \lceil q/2 \rceil$. Isto é, $\bar{a}_i$ é o coeficiente a_i centralizado.

Para facilitar a leitura, quando um polinômio c ou vetor $\mathbf{u}$ tiver coeficientes pequenos, usaremos a cor azul para representá-lo. Note que vetores pequenos tipicamente são secretos, então isso também ajudará a identificar variáveis que representam chaves secretas ou segredos temporários usados durante a encriptação.

1.2.2. Criptografia

O processo de padronização do NIST considera duas primitivas criptográficas importantes, KEMs e assinaturas digitais. Para serem considerados, os esquemas devem propor parâmetros que, dados os melhores ataques conhecidos, instanciem esquemas seguros de sob algum dos 5 critérios definidos pelo NIST.

Os níveis de segurança definidos pelo NIST são apresentados na Tabela 1.1. Para um exemplo concreto, se um KEM apresenta parâmetros de nível de segurança igual a 3, então quebrar[1] o KEM deve ter custo computacional igual ou superior a quebrar uma chave de 192 bits do AES.

Esquemas de encriptação de chave pública: Esquemas de encriptação de chave pública, abreviados por PKE (*public-key encryption*), são definidos por 3 algoritmos: geração de chaves, encriptação e decriptação. A geração de chaves devolve um par de chaves pública e privada, denotadas por **pk** e **sk**, respectivamente. A encriptação recebe uma chave pública **pk** e uma mensagem **m**, e devolve um texto cifrado **c**. O algoritmo de decriptação usa a chave secreta **sk** para obter, a partir do encriptado **c**, a mensagem

[1] Por enquanto, podemos pensar que quebrar significa computar a chave secreta a partir da pública, ou descobrir a chave de sessão que foi encapsulada a partir do texto encriptado e a chave pública.

Tabela 1.1. Níveis de segurança pós-quântica de acordo com o NIST.

Nível de segurança	Dificuldade correspondente
Nível 1	Busca por chave do AES128
Nível 2	Busca por colisão no hash SHA3-256
Nível 3	Busca por chave do AES192
Nível 4	Busca por colisão no hash SHA3-384
Nível 5	Busca por chave do AES256

correspondente.

São considerados dois tipos de atacantes: passivos e ativos. Atacantes passivos não interagem com o portador da chave secreta. Ao atacante passivo, somente é dado acesso à chave pública do alvo, e o atacante pode então usá-la para fazer ataques chamados de CPA (*chosen-plaintext attack*), em que textos legíveis arbitrários são encriptados para tentar detectar padrões que facilitem o ataque. Atacantes ativos são poderosos e podem pedir para o portador da chave secreta decriptar textos encriptados escolhidos, caracterizando um ataque do tipo CCA (*chosen-ciphertext attack*).

Em geral, são considerados 3 tipos de objetivos ao atacar um PKE. Naturalmente, o objetivo mais natural, e devastador, é recuperar a chave secreta **sk** a partir da pública **pk**. Outro objetivo é recuperar o texto legível **m** associado a um encriptado **c** e uma chave pública **pk**. Quando tal recuperação é inviável para um PKE, dizemos que este é unidirecional, abreviado por OW (*one-way*). Há também um objetivo mais fraco, que é, dado um encriptado **c**, a chave pública **pk**, e duas mensagens $\mathbf{m}_0$ e $\mathbf{m}_1$, determinar qual destas duas mensagens foi encriptada para produzir **c**. Um PKE que resiste a este tipo de ataque é dito indistinguível, abreviado por IND.

Um ataque é caracterizado pelo seu objetivo e pelo poder do atacante. Assim, a segurança de um PKE é caracterizada pelo ataque mais poderoso a que ele resiste. Por exemplo, dizemos que um PKE é OW-CPA, quando ele é unidirecional contra ataques passivos. A forte noção de segurança tipicamente objetivada na construção de esquemas é IND-CCA, ou indistinguibilidade de encriptados contra atacantes ativos.

Esquemas de encapsulamento de chaves: Um KEM consiste em 3 algoritmos: geração de chaves, encapsulamento e desencapsulamento. A geração de chaves gera um par de chaves pública e privada, denotadas por **pk** e **sk**, respectivamente. O encapsulamento toma como único parâmetro uma chave pública **pk** e devolve uma chave de sessão **k** e um texto encriptado **c** destinado ao portador da chave secreta associada a **pk**. O desencapsulamento consiste em obter a chave de sessão **k** a partir do texto encriptado **c** usando a chave secreta **sk**. Então, se o desencapsulamento for bem sucedido, o remetente e o destinatário poderão usar a chave **k** num esquema simétrico, como o AES-GCM, para a comunicação segura. As mesmas noções de segurança se aplicam tanto a PKEs quando aos KEMs. Um KEM pode ser construído a partir de um PKE, por exemplo, ao se considerar uma chave de sessão gerada aleatoriamente como a mensagem. Tipicamente, KEMs são construídos sobre PKEs usando o que chamamos de conversões de segurança ou transformações-CCA. Estas transformações pegam um PKE seguro contra atacantes passivos, por exemplo IND-

©2023 Sociedade Brasileira de Computação

Geração de chaves

$\mathbf{pk}_{\text{PKE}}, \mathbf{sk}_{\text{PKE}} \leftarrow$ Gere chaves do PKE
$\sigma \leftarrow$ 256 bits aleatórios
$\mathbf{pk} \leftarrow \mathbf{pk}_{\text{PKE}}$
$\mathbf{sk} \leftarrow (\mathbf{sk}_{\text{PKE}}, \sigma)$
Devolva $(\mathbf{pk}, \mathbf{sk})$

Encapsulamento $(\mathbf{pk})$

$\mathbf{m} \leftarrow$ Mensagem aleatória do PKE
$\mathbf{r} \leftarrow G(\mathbf{m}, \mathbf{pk})$
$\mathbf{c} \leftarrow$ Encripte $(\mathbf{pk}, \mathbf{m}; \mathbf{r})$ com PKE
$\mathbf{k} \leftarrow H(\mathbf{m}, \mathbf{c})$
Devolva $(\mathbf{c}, \mathbf{k})$

Desencapsulamento $(\mathbf{sk} = (\mathbf{sk}_{\text{PKE}}, \sigma), \mathbf{c})$

$\mathbf{k}_{\text{rejeição}} \leftarrow H(\sigma, \mathbf{c})$
$\mathbf{m} \leftarrow$ Decripte$(\mathbf{sk}_{\text{PKE}}, \mathbf{c})$ com PKE
Se $\mathbf{m} = \perp$, **devolva** $\mathbf{k}_{\text{rejeição}}$
$\mathbf{r} \leftarrow G(\mathbf{m}, \mathbf{pk})$
$\hat{\mathbf{c}} \leftarrow$ Encripte $(\mathbf{pk}, \mathbf{m}; \mathbf{r})$ com PKE
Se $\hat{\mathbf{c}} \neq \mathbf{c}$, **devolva** $\mathbf{k}_{\text{rejeição}}$
Devolva $\mathbf{k} \leftarrow H(\mathbf{m}, \mathbf{c})$

Figura 1.1. Transformação de Fujisaki-Okamoto com rejeição implícita para construir um KEM IND-CCA a partir de um PKE (OW ou IND)-CPA.

CPA ou OW-CPA, e produzem um KEM IND-CCA. Ter transformadas desse tipo é muito conveniente, pois é muito mais fácil explicar PKEs na forma OW-CPA ou IND-CPA do que na forma IND-CCA.

A Figura 1.1 mostra uma conversão de segurança muito usada chamada Fujisaki-Okamoto com rejeição implícita [Hofheinz et al. 2017]. A ideia geral tornar inviável para um atacante obter qualquer informação sobre a chave secreta através de pedidos de decriptação. Para isso, a transformação garante propriedades como as seguintes: (i) ser inviável gerar um texto encriptado válido sem conhecer a mensagem em claro associada, pois a aleatoriedade usada na encriptação depende da mensagem; (ii) ser viável a detecção de textos encriptados inválidos durante o desencapsulamento, de modo que o valor $\mathbf{k}_{\text{rejeição}}$ devolvido não comprometa informações secretas.

Esquemas de assinatura: Um esquema de assinatura também consiste em 3 algoritmos: geração de chaves, assinatura e verificação. A geração de chaves é análoga à dos KEMs. O algoritmo de assinatura toma uma mensagem **m** e uma chave secreta **sk**, associada a uma chave pública **pk**, e produz uma assinatura **s**. O algoritmo de verificação toma a chave pública **pk** da entidade que supostamente produziu a assinatura, uma mensagem **m** e a assinatura **s**, e verifica se **s** é de fato uma assinatura válida de **m** sob **pk**. A noção de segurança exigida pelo NIST é chamada de EUF-CMA (*existentially unforgeable with respect to chosen message attacks*). Intuitivamente, um esquema é EUF-CMA se um atacante ativo, que pode pode pedir assinaturas para mensagens arbitrárias, não consegue forjar uma assinatura válida para uma mensagem cuja assinatura válida ele nunca viu.

Hashes, XOFs e geradores pseudoaleatórios: Em geral, as implementações de esquemas criptográficos isolam a aleatoriedade real usada pelas funções num parâmetro chamado `randomness`. Este parâmetro é usado para instanciar geradores pseudo-aleatórios criptograficamente seguros, que então serão usados para a amostragem de valores secretos. Funções de hash criptográficas são muito conhecidas e usadas não só em criptografia, como também em segurança da informação. Estas funções tomam uma sequência de bytes de qualquer tamanho e produzem uma saída com número fixo de bytes. Boas funções de hash para criptografia têm propriedades pseudoaleatórias que garantem, por exemplo, que é intratável encontrar duas cadeias de bytes de mesmo hash. Propriedades como essa fazem tais funções particularmente importantes em sistemas de verificação de integridade,

ou mais recentemente, como provas de trabalho.

Objetos muito convenientes para a geração pseudoaleatória com segurança criptográfica são os XOFs (*extendable-output functions*). Exemplos comumente usados de XOFs são as funções SHAKE128 e SHAKE256. Estas funções funcionam como hashes criptográficos, porém sua saída não tem tamanho fixo. Isso faz com que possamos usá-las em algoritmos genéricos para geração pseudoaleatória.

A biblioteca PyCryptodome[2] implementa XOFs e Hashes em Python. Abaixo mostramos como podemos importar o SHAKE128 para instanciar um XOF com a semente 1234 vista como uma sequência de 32 bytes. Note como o método `read`, que aceita um número de bytes como parâmetro, pode ser usado para obter um novo byte de cada vez.

```
1 sage: xof = SHAKE128.new(int(1234).to_bytes(32))
2 sage: int.from_bytes(xof.read(1))
3 130
4 sage: int.from_bytes(xof.read(1))
5 99
```

Podemos então definir uma coleção de geradores pseudo-aleatórios baseados nos XOFs da seguinte forma. Comece por um gerador de bits.

```
1 def xof_random_bit(xof):
2     byte = int.from_bytes(xof.read(1))
3     return (byte & 1)
```

Também definimos funções mais complexas para gerar inteiros módulo m, isto é entre 0 e $m-1$, ou até entre m_1 e m_2-1.

```
1 MAX_MOD = 2**256
2 MAX_MOD_NBYTES = ceil(log(MAX_MOD, 2**8))
3
4 def xof_random_int_mod(xof, mod):
5     assert mod < MAX_MOD
6     bias_region = MAX_MOD - (MAX_MOD % mod)
7     v = int.from_bytes(xof.read(MAX_MOD_NBYTES))
8     while v >= bias_region:
9         v = int.from_bytes(xof.read(MAX_MOD_NBYTES))
10    return v % mod
11
12 def xof_randrange(xof, min_include, max_exclude):
13     diff = max_exclude - min_include
14     return min_include + xof_random_int_mod(xof, diff)
```

Usando a função `xof_randrange`, podemos implementar o algoritmo de Fisher-Yates para selecionar k itens de uma lista de itens.

```
1 def xof_sample_k_indexes(xof, n_items, k):
2     a = list(range(n_items))
3     for i in range(n_items - 1):
4         j = xof_randrange(xof, i, n_items)
5         a[i], a[j] = a[j], a[i]
6     return a[:k]
```

1.3. NTRU

O NTRU [Chen et al. 2020] é inspirado num esquema de criptografia de chave pública (PKE) introduzido por Hoffstein, Pipher e Silverman em 1998 [Hoffstein et al. 1998].

[2]https://github.com/Legrandin/pycryptodome

Este PKE é baseado em álgebra de polinômios e aritmética modular. Nesta seção, apresentamos a última revisão do NTRU com o conjunto de parâmetros chamado HPS.

Seleção de parâmetros e inicialização: Para cada nível de segurança, o NTRU usa parâmetros (n,q,d,t). Os parâmetros n e q definem o anel polinomial $R_q = \mathbb{Z}_q[x]/\left(x^n-1\right)$ usado como base para as operações. O parâmetro d define o número de coordenadas não nulas nos vetores secretos usados para gerar os polinômios das chaves secretas e textos legíveis, enquanto $t = n-2$ determina seu grau máximo.

Seja $\mathcal{T}$ o conjunto em R_q de grau máximo igual a $t = n-2$, cujos coeficientes estão no conjunto $\{-1,0,1\}$. Considere também o seu subconjunto $\mathcal{T}(d) \subset \mathcal{T}$ que contém polinômios com exatamente $d/2$ coeficientes iguais a -1 e outras $d/2$ coeficientes iguais a 1, onde $d = q/8-2$ é um parâmetro do NTRU.

Para que a decriptação seja possível, é necessário que os parâmetros satisfaçam à seguinte condição. Para quaisquer polinômios $r, f \in \mathcal{T}$ e $g, m \in \mathcal{T}(d)$, o polinômio $\hat{a} = 3rg + fm \in \mathbb{Z}[x]/\left(x^n-1\right)$, deve ter todos os seus coeficientes no intervalo $\{-q/2,\ldots,q/2-1\}$.

Os parâmetros de segurança do NTRU são definidos da seguinte forma:

```
1 @dataclass
2 class NTRUHPS_Parameters:
3     n: int
4     q: int
5     d: int
6     max_degree_t: int
7
8 class NTRU_DPKE_OWCPA:
9     SecurityParameters = {
10        1: NTRUHPS_Parameters(n=509, q=2048, d=254, max_degree_t=509 - 2),
11        3: NTRUHPS_Parameters(n=677, q=2048, d=254, max_degree_t=677 - 2),
12        5: NTRUHPS_Parameters(n=821, q=2048, d=254, max_degree_t=821 - 2),
13    }
```

Considere a fatoração de (x^n-1) em $\Phi_1\Phi_n$, onde $\Phi_1 = (x-1)$ e $\Phi_n = \sum_{i=0}^{n-1} x^i$. Além de R_q, o NTRU usa outros 3 anéis onde certas operações polinomiais são feitas. São eles os anéis $S_q = \mathbb{Z}_q[x]/\Phi_n$, $S_3 = \mathbb{Z}_3[x]/\Phi_n$, e $S_2 = \mathbb{Z}_2[x]/\Phi_n$. O código abaixo mostra a inicialização dos anéis usados pelo NTRU, seguido da inicialização do esquema.

```
1     def __init_rings(self):
2         ring_mod_q = PolynomialRing(Integers(self.params.q), 'x')
3         x = ring_mod_q.gen()
4         phi   = x**self.params.n - 1
5         phi_1 = x - 1
6         phi_n = sum(x**i for i in range(self.params.n))
7         self.Rq = ring_mod_q.quotient_ring(phi)
8         self.Sq = ring_mod_q.quotient_ring(phi_n)
9         self.S3 = ring_mod_q.change_ring(Integers(3)).quotient(phi_n)
10        self.S2 = ring_mod_q.change_ring(Integers(2)).quotient(phi_n)
11
12    def __init__(self, security_level):
13        self.params = self.SecurityParameters[security_level]
14        self.__init_rings()
```

Para o funcionamento do NTRU, é necessário converter polinômios entre os anéis. Suponha que queremos converter um polinômio f em $R_1 = \mathbb{Z}_{q_1}/m_1(x)$ para um polinômio noutro anel $R_2 = \mathbb{Z}_{q_2}/m_2(x)$. Então, primeiro escrevemos o polinômio com coeficientes

Geração de chaves	**Encriptação** $(\mathbf{pk}, (r, m) \in \mathcal{T} \times \mathcal{T}(d))$	**Decriptação** $(\mathbf{sk}, c)$
$f \leftarrow$ **Uniforme** $(\mathcal{T})$	$h \leftarrow \mathbf{pk}$	**Se** $R_q(c) \neq 0 \bmod \Phi_1$, **devolva** $\perp$
$g \leftarrow$ **Uniforme** $(\mathcal{T}(d))$	$c \leftarrow R_q(h)R_q(r) + R_q(m)$	$f, S_3(f)^{-1}, S_q(h)^{-1} \leftarrow \mathbf{sk}$
$h \leftarrow 3R_q(g)R_q\left(S_q(f)^{-1}\right)$	**Devolva** c	$a \leftarrow S_q(c)S_q(f)$
$\mathbf{pk} \leftarrow h$		$m \leftarrow S_3(a)S_3(f)^{-1}$
$\mathbf{sk} \leftarrow \left(f, S_3(f)^{-1}, S_q(h)^{-1}\right)$		$r \leftarrow S_q\left(R_q(c) - R_q(m)\right)S_q(h)^{-1}$
Devolva $(\mathbf{pk}, \mathbf{sk})$		**Se** $r, m \notin \mathcal{T} \times \mathcal{T}(d)$, **devolva** $\perp$
		Devolva (r, m)

Figura 1.2. PKE determinístico do NTRU.

centralizados $f = f_0 + f_1x + \ldots + f_nx^n$, onde cada f_i está no intervalo $-q_1/2 \leq f_i < q_1/2$. Depois computamos o seguinte polinômio convertido, denotado por $R_2(f)$, como $R_2(f) = (f_0 \bmod q_2) + \ldots + (f_n \bmod q_2)x^n \mod m_2(x)$.

Note que, como o módulo $\Phi = \Phi_1\Phi_n$ de R_q é um múltiplo dos módulos Φ_n usado em S_q, então: $S_q\left(R_q(a) + R_q(b)R_q(c)\right) = S_q(a) + S_q(b)S_q(c)$. No entanto, esta propriedade não vale no outro sentido, isto é, em geral $R_q(S_q(a) + S_q(b)) \neq R_q(a) + R_q(b)$.

A operação de centralização dos coeficientes pode ser escrita da seguinte forma:

```
1   def centered(self, polynomial):
2       modulo = polynomial.base_ring().order()
3
4       def mod_centered(v):
5           v = int(v) % modulo
6           if v <= modulo // 2:
7               return v
8           return - (modulo - v)
9
10      return [mod_centered(c) for c in polynomial]
```

Assim, a conversão de um polinômio para outro anel é dada por:

```
1   def to_Rq(self, polynomial):
2       return self.Rq(self.centered(polynomial))
3   def to_Sq(self, polynomial):
4       return self.Sq(self.centered(polynomial))
5   def to_S3(self, polynomial):
6       return self.S3(self.centered(polynomial))
7   def to_S2(self, polynomial):
8       return self.S2(self.centered(polynomial))
```

A Figura 1.2 mostra os algoritmos usados no NTRU. Nas próximas seções, vamos descrever cada uma delas com a sua respectiva implementação.

Geração de chaves: A geração de chaves é implementada da seguinte forma:

```
1   def keygen(self, randomness):
2       xof = SHAKE256.new(randomness)
3       f_in_s3 = self.S3(self.gen_ternary_coeffs(xof))
4       g_in_s3 = self.S3(self.gen_ternary_coeffs_for_d(xof, self.params.d))
5       f_inv_3 = f_in_s3.inverse()
6       f = self.to_Rq(f_in_s3)
7       g = self.to_Rq(g_in_s3)
8       h = 3 * g * self.to_Rq(self.Sq_inverse(f))
9       h_inv_q = self.to_Rq(self.Sq_inverse(h))
10      return h, (f, f_inv_3, h_inv_q)
```

Primeiro, polinômios f e g são escolhidos de $\mathcal{T}$ e $\mathcal{T}(d)$, respectivamente, usando a XOF e as funções apropriadas. Os coeficientes de polinômios em $\mathcal{T}$ e $\mathcal{T}(d)$ são gerados usando o código abaixo:

```
1    def gen_ternary_coeffs(self, xof):
2        return [choice([-1, 0, 1]) for _ in range(self.params.max_degree_t + 1)]
3
4    def gen_ternary_coeffs_for_d(self, xof, d):
5        assert d % 2 == 0
6        coeffs = [0] * (self.params.max_degree_t + 1)
7        supp = xof_sample_k_indexes(xof, self.params.max_degree_t + 1, d)
8        for i in range(d // 2):
9            coeffs[supp[i]] = 1
10           coeffs[supp[d//2 + i]] = -1
11       return coeffs
```

Note que, como $f \in \mathcal{T}$, então 2 não divide f, e portanto f admite inversa em S_q. Além disso, lembre que S_2 e S_3 são corpos, então quaisquer de seus elementos não nulos admitem inversos, inclusive f.

Nos corpos S_2 e S_3, calcular a inversa é fácil através do algoritmo de Euclides estendido, e o Sage as calcula simplesmente através do método `inverse`. Porém, para S_q que não é um corpo, vamos implementar uma função eficiente para o cálculo da inversa customizada para este caso. A função `Sq_inverse` calcula o inverso de um polinômio $a \in S_q$, supondo que existe. Seu código é dado abaixo, e em seguida explicamos por que ela funciona.

```
1    def Sq_inverse(self, a):
2        assert is_power_of_two(self.params.q)
3        a_in_Sq = self.to_Sq(a)
4        v0 = self.to_Sq(self.to_S2(a).inverse())
5        t = 1
6        while t < log(self.params.q, 2):
7            v0 = v0 * (2 - a_in_Sq * v0)
8            t = 2*t
9        assert (v0 * a_in_Sq) == 1
10       return v0
```

A ideia da função é começar com v_0 sendo o inverso de a em S_2 e usar este inverso para construir inversos em $S_{2^2}, S_{2^4}, S_{2^8}, \ldots$, progressivamente, onde $S_i = \mathbb{Z}_i[x]/(\Phi_n)$. O critério de parada garante que sejam feitas pelo menos $\log_2(\log_2 q)$ iterações, que são suficientes para que o inverso em S_q seja encontrado.

Vamos analisar a primeira iteração. Para isso, considere $v_1 = v_0(2 - av_0)$. Pela definição de v_0, sabemos que $av_0 \equiv 1 \bmod 2$, então $av_0 = 2m+1$, para algum polinômio m. Mas isso implica que: $av_1 = av_0(2-av_0) = (2m+1)(2-2m-1) = (2m+1)(-2m+1)$. Portanto $av_0 = -4m^2+1 \equiv 1 \bmod 4$, isto é, v_1 é o inverso de a em S_4. Este argumento pode ser iterado para mostrar, por indução, que o resultado de `Sq_inverse` de fato é uma inversa em S_q.

Encriptação: O texto legível do NTRU é um par (r,m) em que $r \in \mathcal{T}$ e $m \in \mathcal{T}(d)$. Implementamos abaixo uma função que gera um texto legível (r,m) através de bytes usados como semente em um XOF.

```
1    def sample_plaintext(self, randomness):
2        xof = SHAKE256.new(randomness)
3        r = self.gen_ternary_coeffs(xof)
```

©2023 Sociedade Brasileira de Computação

```
4         m = self.gen_ternary_coeffs_for_d(xof, self.params.d)
5         return r, m
```

A encriptação é simples, basta computar $c = hr + m$ com as operações em R_q.

```
1     def encrypt(self, pk, plaintext):
2         h = pk
3         r_coeffs, m_coeffs = plaintext
4         r = self.Rq(r_coeffs)
5         m = self.Rq(m_coeffs)
6         c = h * r + m
7         return c
```

Decriptação: Abaixo, listamos o código da decriptação:

```
1     def decrypt(self, sk, ciphertext):
2         (f, f_inv_3, h_inv_q) = sk
3         c = ciphertext
4         if sum(c) != 0:
5             return None
6         a = (self.to_Sq(c) * self.to_Sq(f))
7         m = self.to_S3(a) * f_inv_3
8         r = (c - self.to_Rq(m)) * h_inv_q
9         m = self.centered(m)
10        r = self.centered(self.to_Sq(r))
11        if self.is_valid_plaintext(r, m):
12            return r, m
13        else:
14            return None
```

A decriptação começa com um teste de validade do texto encriptado c. Se c foi calculado corretamente, então $c = hr + m$, e h é um polinômio múltiplo de g. Como tanto m quando g foram selecionados de $m, g \in \mathcal{T}(d)$, então ambos os polinômios têm mesmo número de coeficientes iguais a 1 e -1.

Dessa forma, $m(1) = g(1) = 0 \bmod q$, o que implica que $c(1) = h(1)r(1) + m(1) = 0 \bmod q$. Implementamos a computação de $c(1)$ como a soma de seus coeficientes. Note que, em Sage, não precisamos colocar o módulo q pois as operações sobre coeficientes de c já são feitas em $\mathbb{Z}_q$.

Vamos agora verificar que a decriptação funciona. O polinômio a calculado durante a decriptação é

$$\begin{aligned} a &= S_q(c)\, S_q(f) \\ &= S_q\left(R_q(h)\, R_q(r) + R_q(m)\right) S_q(f) \\ &= S_q(h)\, S_q(r)\, S_q(f) + S_q(m)\, S_q(f)\,. \end{aligned}$$

Considere a primeira parcela

$$\begin{aligned} S_q(h) S_q(f)\, S_q(r) &= S_q\left(3R_q(g)\, R_q\left(S_q(f)^{-1}\right)\right) S_q(f)\, S_q(r) \\ &= 3S_q(g)\, S_q(f)^{-1}\, S_q(f)\, S_q(r) \\ &= 3S_q(g)\, S_q(r)\,. \end{aligned}$$

Então $a = 3S_q(g)S_q(r) + S_q(m)S_q(f)$. Mas lembre que os parâmetros do NTRU são selecionados de forma que $\hat{a} = 3gr + mf \in \mathbb{Z}[x]/(x^n - 1)$ tenha sempre coeficientes no intervalo $\{-q/2, \ldots, q/2-1\}$. Então $\hat{a}$ mod $3 = S_3(\hat{a}) = S_3(a) = S_3(m)S_3(f)$. Isso implica que $S_3(a)S_3(f)^{-1} = S_3(m)S_3(f)S_3(f)^{-1} = S_3(m)$. Basta então centralizar os coeficientes de $S_3(m)$ para obter o valor de m original com coeficientes em $\{-1, 0, 1\}$.

Agora que temos m, podemos recuperar a segunda parte r, da seguinte forma: $S_q\left(R_q(c) - R_q(m)\right)S_q(h)^{-1} = S_q(h)S_q(r)S_q(h)^{-1} = S_q(r)$.

Para garantir que a decriptação foi correta, verifica-se se de fato $r \in \mathcal{T}$ e $m \in \mathcal{T}(d)$. Esta verificação é implementada pela seguinte função.

```
1    def is_valid_plaintext(self, r, m):
2        if (m.count(-1), m.count(1)) != (self.params.d//2, self.params.d//2):
3            return False
4        r_non_ternary_coefficients = len([c for c in r if c not in {-1, 0, 1}])
5        if r_non_ternary_coefficients:
6            return False
7        return True
```

1.4. Fundamentos do Kyber e Dilithium: NTT e módulos

Nesta seção, vamos discutir conceitos fundamentais para os esquemas da família Crystals, isto é, o Kyber e o Dilithium. Começamos discutindo a Transformada da Teoria dos Números (NTT), que é parte essencial de ambos os esquemas. Depois discutimos módulos, que são uma generalização de espaços vetoriais, e discutimos a sua implementação.

1.4.1. Multiplicação rápida de polinômios com a NTT

A multiplicação de polinômios é uma operação recorrente no contexto dos esquemas criptográficos apresentados neste minicurso. Então realizar essa operação de forma eficiente é fundamental para atingir desempenho adequado. Nesta seção, vamos apresentar a NTT, uma transformada da família da transformada de Fourier, que é usada tanto pelo Kyber quanto pelo Dilithium para fazer as multiplicações de polinômios.

A apresentação é dividida em três partes. Primeiro mostramos como o Teorema Chinês do Resto pode ajudar na multiplicação de polinômios. Depois mostramos o anel polinomial usado pelo Kyber e pelo Dilithium e como este pode ser quebrado em anéis menores onde as contas são mais eficientes. Finalmente, apresentamos a implementação rápida da NTT.

Teorema Chinês do Resto para polinômios: Fixe um primo q, e um polinômio $m(x)$ de grau n. Considere o anel polinomial $R_q = \mathbb{F}_q[x]/m(x)$ dos resíduos módulo m, isto é

$$\mathbb{F}_q[x]/m(x) = \left\{a \bmod m : a \in \mathbb{F}_q[x]\right\}.$$

Dados dois polinômios a e b de R_q, queremos computar o produto $c = ab$ de forma eficiente. Lembre que, como $a, b \in R_q$, ambos os polinômios têm grau menor que n. Além disso, como o produto $c = ab$ também tem grau menor que 256 pois, pois todas as operações em R_q são feitas $\bmod m(x)$.

©2023 Sociedade Brasileira de Computação

Suponha que exista uma fatoração de $m(x)$ em s polinômios $f_1, \ldots, f_s$, isto é

$$m(x) = \prod_{i=1}^{s} f_i.$$

Além disso, suponha que os polinômios f_i são coprimos entre si, isto é $\gcd(f_i, f_j) = 1$ para $i \neq j$.

Então, o Teorema Chinês do resto garante que existe apenas um polinômio $a \mod m(x)$ que satisfaz o seguinte sistema de equações modulares

$$\begin{cases} a & = & a_1 & \mod f_1, \\ \vdots & \vdots & \vdots & \vdots \\ a & = & a_s & \mod f_s. \end{cases}$$

Dessa forma, se soubermos os resíduos dos polinômios a e b em relação a cada um dos polinômios f_i, podemos multiplicar os resíduos correspondentes e usar o Teorema Chinês do Resto para obter o produto módulo $m(x)$.

Há, porém, duas observações importantes. A primeira é que se os fatores f_i tiverem grau muito alto, a multiplicação entre resíduos será ineficiente. A segunda é que precisamos ser capazes de computar eficientemente tanto os resíduos a_i e b_i quanto o polinômio c a partir dos resíduos $a_i b_i \mod f_i$.

Fatorando polinômios com raízes da unidade: Vimos que é desejável que o polinômio m seja fatorado em polinômios de grau pequeno para que a multiplicação de resíduos seja rápida. Vamos considerar um caso que é usado tanto pelo Kyber como pelo Dilithium: $m(x) = x^n + 1$, onde n é uma potência de 2.

Fixe um primo q para o qual $\mathbb{F}_q$ admite uma raiz primitiva ω de ordem $2n$ da unidade, isto é $\omega^{2n} = 1$, e não existe nenhum outro $0 < i < 2n$ tal que $\omega^i = 1$. Podemos então fatorar $m(x) = x^n + 1$ notando que $\omega^n = -1$. Escreva $m(x) = x^n - \omega^n$, e note que m então é a diferença de dois quadrados, podendo ser fatorado em

$$m(x) = \left(x^{(n/2)} - \omega^{(n/2)}\right)\left(x^{(n/2)} + \omega^{(n/2)}\right).$$

Se $n/2 = 1$, então a fatoração está completa. Suponha então que n é uma potência de 2 maior que 1. Note que podemos fatorar novamente $\left(x^{(n/2)} - \omega^{(n/2)}\right)$ usando a mesma observação. Considere então como fatorar $\left(x^{(n/2)} + \omega^{(n/2)}\right)$. Mas como $\omega^n = -1$, podemos transformar este fator em outra diferença de dois quadrados escrevendo

$$x^{(n/2)} + \omega^{(n/2)} = x^{(n/2)} - (\omega^n)\omega^{(n/2)} = x^{(n/2)} - \omega^{n+(n/2)}.$$

Usando este procedimento recursivamente $\log_2(n)$ vezes, é possível chegar até a fatoração em polinômios de grau 1.

©2023 Sociedade Brasileira de Computação

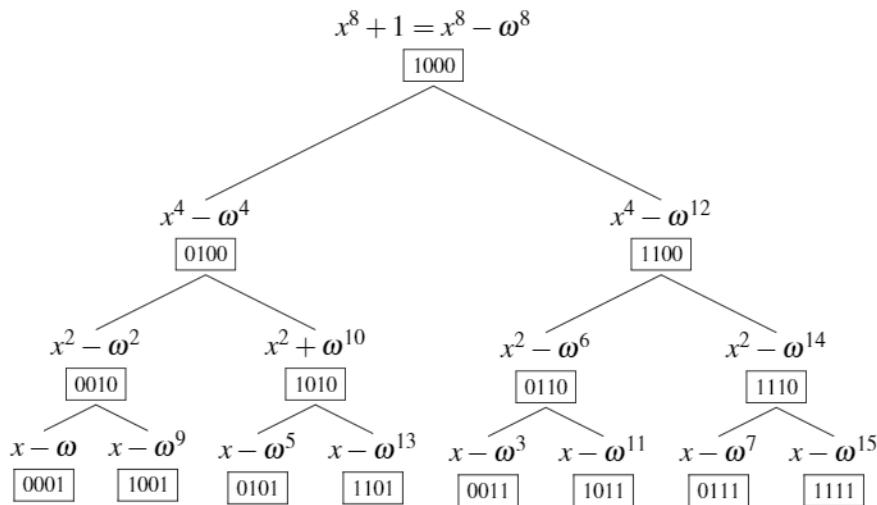

Figura 1.3. Decomposição de x^8+1 usando uma raiz da unidade ω de ordem 16.

A Figura 1.3 ilustra a árvore de decomposição para o polinômio x^8+1, supondo que existe uma raiz primitiva da unidade de ordem 16 em $\mathbb{F}_q$. As etiquetas abaixo dos polinômios indicam a representação em binário da potência de ω correspondente.

A ordem em que as potências de ω aparecem pode parecer misteriosa a princípio. Porém podemos notar um padrão interessante nas etiquetas usadas na Figura 1.3: o filho do nó de etiqueta a têm etiquetas $a/2$ e $a/2+8$. Isso faz com que a ordem em que as etiquetas são usadas na árvore, lendo linha por linha, seja igual à ordem das sequências em binário usando 4 bits, porém considerando a ordem invertida. Isto é 1000, 0100, 1100, e assim por diante. Essa ordem é chamada de bit-invertida, e pode ser implementada em Sage através da seguinte função.

```
def bit_rev(x, length):
    b = bin(x)[2:]
    b = '0' * (length - len(b)) + b
    return int(''.join(reversed(b)), 2)
```

Note que, apesar de termos mostrado a árvore de decomposição completa em polinômios de grau 1, existe também o caso incompleto. Quando n é uma potência de 2 e considerando o polinômio $m=1$, o caso incompleto ocorre quando ω é uma raiz primitiva de ordem 2° em F_q, tal que $2 \leq o \leq \log_2 n$. Nesse caso, o mesmo procedimento é usado para decompor x^n+1, porém, a decomposição acaba com polinômios de grau $\log_2 n - o + 1$.

Sabemos então como construir polinômios x^n+1 que podem ser fatorados em polinômios de grau baixo, que são convenientes para a multiplicação baseada no Teorema Chinês do Resto. Porém, para que seja possível fazer a multiplicação eficiente entre dois polinômios a e b módulo x^n+1, é preciso definir um algoritmo rápido para fazer conversão a rápida dos polinômios a e b em resíduos em relação aos fatores, e vice-versa.

NTT: computação rápida dos resíduos na árvore de decomposição: Nesta seção, apresentamos a transformada da Teoria dos Números, do inglês Number Theory Transform (NTT), que consiste numa variação discreta da transformada Rápida de Fourier

©2023 Sociedade Brasileira de Computação

(FFT). Como a FFT, a NTT pode ser calculada eficientemente. O que nos faz particularmente interessados na NTT em nossa discussão é que, com ela, é possível calcular precisamente os resíduos de um polinômio a numa árvore de decomposição de x^n+1 em F_q, quando n é uma potência de 2.

Uma forma direta de implementar a NTT de um polinômio a num anel $R_q = \mathbb{F}_q[x]/(x^n+1)$ seria a seguinte. Suponha que temos uma raiz da unidade ω de ordem $2n$. Comece na raiz da árvore de decomposição de (x^n+1). Então compute $a_0 = a \bmod x^n - \omega^n$ e $a_1 = a \bmod x^n + \omega^n$, e prossiga recursivamente nas subárvores de raízes $x^n - \omega^n$ e $x^n + \omega^n$, e polinômios a_0 e a_1, respectivamente.

Porém uma forma mais rápida de implementá-la é adaptando os algoritmos da FFT, como os de Cooley-Tukey e de Gentleman-Sande, para usar a nossa raiz da unidade num corpo finito.

Vamos mostrar a implementação da NTT rápida para o caso específico em que $n = 256$, e para o caso de $R_q = \mathbb{F}_q/(x^n+1)$. Em inglês, é comum chamar a NTT no anel $R_q = \mathbb{F}_q/(x^n+1)$ de *negacyclic* NTT pois, $ax^n = -a \bmod (x^n+1)$.

```
1  def ntt_leveled_negacyclic_256(a, field, omega, n_levels):
2      assert len(a) == 256
3      omega = field(omega)
4      assert omega.multiplicative_order() == 2**(n_levels + 1)
5
6      a_hat = copy(a)
7      for l in range(7, 7 - n_levels, -1):
8          for i in range(0, 2**(7 - l)):
9              phi = omega ** bit_rev(2**(7 - l) + i, n_levels)
10             for j in range(i * 2**(l + 1), i * 2**(l + 1) + 2**l):
11                 t0 = a_hat[j]
12                 t1 = phi * a_hat[j + 2**l]
13                 a_hat[j] = t0 + t1
14                 a_hat[j + 2**l] = t0 - t1
15     return vector(field, a_hat)
```

Similarmente, a inversa da NTT pode ser calculada da seguinte forma. Note que, de forma análoga à DFT, é necessário normalizar o resultado.

```
1  def inv_ntt_leveled_negacyclic_256(a_hat, field, omega, n_levels):
2      assert len(a_hat) == 256
3      omega = field(omega)
4      assert omega.multiplicative_order() == 2**(n_levels + 1)
5
6      a = copy(a_hat)
7      for l in range(8 - n_levels, 8):
8          for i in range(2**(7 - l)):
9              phi = omega ** -bit_rev(2**(7 - l) + i, n_levels)
10             for j in range(i * 2**(l + 1), i * 2**(l + 1) + 2**l):
11                 t0 = a[j] + a[j + 2**l]
12                 t1 = a[j] - a[j + 2**l]
13                 a[j] = t0
14                 a[j + 2**l] = phi * t1
15     return vector(field, a) * field(2**n_levels)**(-1)
```

Vejamos como usar a NTT para multiplicar dois polinômios. Suponha que estamos em um anel que admite a decomposição completa. Para um exemplo concreto, tome o caso do Dilithium em que $q = 8380417$ e $\omega = 1753$. Como este caso a NTT rápida é completa, a multiplicação no domínio da NTT corresponde à multiplicação de coeficientes em $\mathbb{F}_q$.

©2023 Sociedade Brasileira de Computação

Podemos usar o seguinte código para verificar que a multiplicação usando a NTT de fato dá o mesmo resultado que a multiplicação em Sage no anel R_q.

```
F = GF(8380417)
PolyRing = PolynomialRing(F, 'x')
x = PolyRing.gen()
Rq = PolyRing.quotient(x**256 + 1)
omega = F(1753)

def poly_prod_negacyclic_256_complete(a, b):
    a_ntt = ntt_leveled_negacyclic_256(a, F, omega, 8)
    b_ntt = ntt_leveled_negacyclic_256(b, F, omega, 8)
    c_ntt = [a_i * b_i for a_i, b_i in zip(a_ntt, b_ntt)]
    return inv_ntt_leveled_negacyclic_256(c_ntt, F, omega, 8)

def poly_prod_in_sage(a, b):
    return vector(Rq(list(a)) * Rq(list(b)))

a = random_vector(F, 256)
b = random_vector(F, 256)

result_ntt_mult = poly_prod_negacyclic_256_complete(a, b)
result_sage_mult = poly_prod_in_sage(a, b)
print(f'{(result_ntt_mult == result_sage_mult) = }')
```

1.4.2. Módulos

Módulos são generalizações de espaços vetoriais em que os escalares são elementos de um anel, e não necessariamente de um corpo finito. Essas estruturas são convenientes, pois permitem que trabalhemos com vetores e matrizes de polinômios, usando as regras e notações convencionais de álgebra linear, como a adição de vetores, multiplicação por escalares e produto escalar.

O conceito pode ser facilmente entendido com um exemplo. Para isso, consideremos o anel polinomial usado pelo Dilithium definido a seguir. Seja o primo $q = 8380417$, e considere o anel polinomial $R_q = \mathbb{F}_q[x]/\left(x^{256}+1\right)$. Para qualquer inteiro $k \geq 1$, o conjunto R_q^k

$$R_q^k = \left\{ \begin{bmatrix} p_1 \\ \vdots \\ p_k \end{bmatrix} : p_i \in R_q \right\},$$

é um módulo de dimensão k sobre R_q.

Os elementos dos módulos, que chamaremos de vetores, podem ser convenientemente representados pela classe `PolynomialVector` abaixo. Note como, para instanciar um módulo, devemos passar um anel onde a NTT pode ser computada eficientemente, além de uma lista de vetores de coeficientes dos polinômios e uma bandeira que indica se o vetor está no domínio da NTT ou não.

```
class PolynomialVector():

    def __init__(self, ntt_ring, poly_vec_coefficients, in_ntt_domain=False):
        assert all(len(p) == ntt_ring.n for p in poly_vec_coefficients)
        self.ntt_ring = ntt_ring
        self.poly_vec = [vector(ntt_ring.field, p) for p in poly_vec_coefficients]
        self.in_ntt_domain = in_ntt_domain
```

Podemos usar os métodos mágicos de Python para permitir que as operações como adição, subtração e negação sejam representadas de forma concisa no código através dos

respectivos operadores. Note que, para evitar problemas de domínio, temos um método que testa se dois vetores de polinômio são compatíveis.

```
1
2     def test_compatibility_of_domains(self, poly_vec2):
3         assert self.ntt_ring.field == poly_vec2.ntt_ring.field
4         assert self.in_ntt_domain == poly_vec2.in_ntt_domain
5
6     def __add__(self, poly_vec2):
7         self.test_compatibility_of_domains(poly_vec2)
8         sum_poly_vec = [v1 + v2 for v1, v2 in zip(self.poly_vec, poly_vec2.poly_vec)]
9         return PolynomialVector(self.ntt_ring, sum_poly_vec, in_ntt_domain=self.
    in_ntt_domain)
10
11    def __sub__(self, poly_vec2):
12        self.test_compatibility_of_domains(poly_vec2)
13        sum_poly_vec = [v1 - v2 for v1, v2 in zip(self.poly_vec, poly_vec2.poly_vec)]
14        return PolynomialVector(self.ntt_ring, sum_poly_vec)
15
16    def __neg__(self):
17        return PolynomialVector(self.ntt_ring, [-p for p in self.poly_vec])
```

Note como o cálculo da NTT em cada polinômio é delegado ao anel usado para a inicialização.

```
1     def ntt(self):
2         poly_vec_ntt = [self.ntt_ring.ntt(p) for p in self]
3         return PolynomialVector(self.ntt_ring, poly_vec_ntt, in_ntt_domain=True)
4
5     def inv_ntt(self):
6         poly_vec = [self.ntt_ring.inv_ntt(p) for p in self]
7         return PolynomialVector(self.ntt_ring, poly_vec, in_ntt_domain=False)
```

Através de uma implementação polimórfica da multiplicação, podemos usar o mesmo operador $*$ para calcular: *(i)* o produtos de vetores de polinômios por inteiros, *(ii)* o produtos de vetores de polinômios por polinômios, e *(iii)* o produto escalar entre dois vetores de polinômios.

```
1     def _multiply_poly_vecs(self, poly_vec2):
2         self.test_compatibility_of_domains(poly_vec2)
3
4         if (self.in_ntt_domain == True):
5             prod = sum(self.ntt_ring.poly_product_ntt_domain(p1, p2)
6                        for p1, p2 in zip(self, poly_vec2))
7             return prod
8
9         else:
10            raise NotImplementedError
11
12    def _multiply_poly_vec_and_poly(self, poly):
13        assert self.in_ntt_domain
14
15        prod = [self.ntt_ring.poly_product_ntt_domain(poly, p) for p in self]
16        return PolynomialVector(self.ntt_ring, prod, in_ntt_domain=True)
17
18    def _multiply_poly_vec_by_scalar(self, field_element):
19        return PolynomialVector(self.ntt_ring, [p * field_element for p in self], self.
    in_ntt_domain)
20
21    def __mul__(self, poly_or_poly_vec_or_int):
22        if isinstance(poly_or_poly_vec_or_int, PolynomialVector):
23            return self._multiply_poly_vecs(poly_or_poly_vec_or_int)
24        elif poly_or_poly_vec_or_int in self.ntt_ring.field:
25            return self._multiply_poly_vec_by_scalar(poly_or_poly_vec_or_int)
26        else:
27            return self._multiply_poly_vec_and_poly(poly_or_poly_vec_or_int)
```

1.5. Crystals-Kyber

O Kyber [Avanzi et al. 2019] é um esquema de encapsulamento de chaves baseado no problema (*Learning With Errors*) em módulos (MLWE). Começamos a seção definido o anel usado pelo Kyber para em seguida definir o MLWE. Depois apresentamos um esquema de chave pública geral baseado no MLWE, sobre o qual o Kyber é construído. Ao final, o Kyber e seus algoritmos são apresentados.

1.5.1. O anel polinomial usado pelo Kyber

Em todos os seus níveis de segurança, o Kyber usa o primo $q = 3329$ e o anel polinomial $R_q = \mathbb{F}_q/(x^n+1)$ para $n = 256$. Uma característica importante deste anel é que ele não admite raiz primitiva da unidade de ordem 512, que sabemos ser necessária para aplicar a NTT negacíclica rápida completa quando $n = 256$. Porém, $\omega = 17$ é uma raiz de ordem 256 da unidade em R_q, então podemos calcular a NTT rápida de 7 níveis, obtendo a decomposição de (x^n+1) como

$$x^{256}+1 = \prod_{i=0}^{127}\left(x^2-\omega^{2i+1}\right) = \prod_{i=0}^{127}\left(x^2-\omega^{2\texttt{bit_rev}(i,7)+1}\right).$$

Representamos o anel R_q usado pelo Kyber pela seguinte classe.

```
1  class KyberNTTRing():
2  
3      def __init__(self):
4          self.field = GF(3329)
5          self.n = 256
6          self.omega_n = self.field(17)
7          self.polynomial_ring = PolynomialRing(self.field, 'x')
8          self.x = self.polynomial_ring.gen()
9  
10     def ntt(self, a):
11         return ntt_leveled_negacyclic_256(a, self.field, self.omega_n, 7)
12 
13     def inv_ntt(self, a_hat):
14         return inv_ntt_leveled_negacyclic_256(a_hat, self.field, self.omega_n, 7)
```

Como a NTT e sua inversa são computadas de forma incompleta em 7 níveis, a multiplicação no domínio da NTT deve ser feita módulo cada um dos fatores $x^2 - \omega^{2\texttt{bit_rev}(i,7)+1}$.

```
1      def poly_product_ntt_domain(self, a_hat, b_hat):
2  
3          prod = [None] * 256
4          for i in range(0, 256, 2):
5              pa = a_hat[i + 1]*self.x + a_hat[i]
6              pb = b_hat[i + 1]*self.x + b_hat[i]
7  
8              pol_prod = (pa * pb).mod(self.x**2 - self.omega_n**(2*bit_rev(i//2, 7) + 1))
9              prod[i + 1] = pol_prod[1]
10             prod[i] = pol_prod[0]
11 
12         return vector(self.field, prod)
```

1.5.2. O problema MLWE

Em alto nível, o MLWE consiste em encontrar o vetor de um módulo que é a solução de um sistema linear corrompido por erros pequenos. Então, antes de definir formalmente o MLWE, vamos apresentar a distribuição de erros usada.

©2023 Sociedade Brasileira de Computação

Definição 1 (Distribuição binomial centrada)**.** Denotamos por $\mathcal{B}_\eta$ a binomial centrada no 0 com parâmetros $(n = 2\eta, p = 1/2)$. Para amostrar um valor b da distribuição $\mathcal{B}_\eta$, escolha 2η bits $b_1, \ldots, b_{2\eta}$ de forma uniformemente aleatória, e devolva

$$b = \sum_{i=1}^{\eta} b_i - \sum_{i=\eta}^{2\eta} b_i = \sum_{i=1}^{\eta} (b_i - b_{i+\eta}).$$

□

Podemos implementar a amostragem de $\mathcal{B}_\eta$ usando um XOF já inicializado da seguinte forma.

```
1 def xof_centered_binomial_sample(xof, eta):
2     a = sum(xof_random_bit(xof) for _ in range(eta))
3     b = sum(xof_random_bit(xof) for _ in range(eta))
4     return a - b
```

Denotamos por $\mathcal{B}_\eta(R_q)$ a distribuição sobre o anel polinomial R_q em que o coeficiente de cada polinômio é selecionado independentemente segundo a distribuição $\mathcal{B}_\eta$. Similarmente, usamos $\mathcal{B}_\eta(R_q^k)$ para denotar a distribuição sobre vetores de polinômios em que cada polinômio é escolhido segundo a distribuição $\mathcal{B}_\eta(R_q)$.

Definição 2 (Problema MLWE computacional)**.** Sorteie uma matriz $\mathbf{A}$ aleatoriamente de forma uniforme de $R_q^{m\times k}$. Escolha um vetor $\mathbf{s} \in R_q^k$ segundo a distribuição $\mathcal{B}_\eta(R_q^k)$ e outro vetor $\mathbf{e} \in R_q^m$ de acordo com $\mathcal{B}_\eta(R_q^m)$. Compute $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}$. O problema MLWE computacional de parâmetros $(m, k, \mathcal{B}_\eta)$ consiste em, dados $\mathbf{A}$ e $\mathbf{b}$, encontrar o vetor $\mathbf{s}$.

□

Definição 3 (Problema MLWE decisional)**.** $\mathbf{A} \in R_q^{m\times k}$ e um vetor $\mathbf{b} \in R_q^m$. O problema MLWE decisional de parâmetros $(m, k, \mathcal{B}_\eta)$ consiste em distinguir entre os dois casos possíveis.

1. $(\mathbf{A}, \mathbf{b})$ foram tirados de forma uniforme dos conjuntos $R_q^{m\times k}$ e R_q^m, respectivamente.
2. $\mathbf{A}$ foi escolhida de forma uniforme de $R_q^{m\times k}$ mas $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e}$, onde $\mathbf{s}$ e $\mathbf{e}$ foram escolhidos segundo $\mathcal{B}(R_q^k)$ e $\mathcal{B}(R_q^m)$, respectivamente.

□

1.5.3. Um protótipo de esquema criptográfico baseado no MLWE

Suponha que o MLWE é intratável em sua versão de decisão, para o anel R_q. Pode-se argumentar que a ideia mais natural para usá-lo para definir um esquema criptográfico começa com usar vetores $(\mathbf{s}, \mathbf{e})$ como a chave secreta e o par $(\mathbf{A}, \mathbf{t})$ como a chave pública, e dessa forma seria intratável obter a chave secreta a partir da chave pública.

Geração de chaves: Escolha dois vetores $\mathbf{s}$ e $\mathbf{e}$ de acordo com as distribuições $\mathcal{B}_\eta(R_q^k)$, respectivamente. Sorteie uma matriz $\mathbf{A}$, de dimensões $k \times k$, formada por polinômios de R_q selecionados aleatoriamente. Compute $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$. Então as chaves pública e privada são os pares $(\mathbf{A}, \mathbf{t})$ e $(\mathbf{s}, \mathbf{e})$, respectivamente.

Encriptação: Dada uma chave pública $(\mathbf{A}, \mathbf{t})$, a ideia da encriptação é criar instâncias do MLWE a partir de $\mathbf{A}$ e $\mathbf{t}$ que, quando combinadas com a chave secreta, permitem que a mensagem seja recuperada.

Seja $\mathbf{m}$ uma mensagem binária que queremos encriptar. Denote por $\text{Encode}(\mathbf{m})$ a codificação de $\mathbf{m}$ num polinômio de R_q da seguinte forma:

$$\text{Encode}(\mathbf{m})[i] = \begin{cases} 0, & \text{se } \mathbf{m}[i] = 0, \text{ e} \\ \lceil q/2 \rceil, & \text{se } \mathbf{m}[i] = 1. \end{cases}$$

A ideia da codificação é que seja possível recuperar $\mathbf{m}$ mesmo que o polinômio codificado sofra corrupções de tamanho pequeno em relação a q. A decodificação de um polinômio c é feita por $\text{Decode}(c)$ que verifica se cada coeficiente do polinômio c está mais próximo de 0 ou de $q/2 \mod q$, e decodifica para o bit correspondente. Na classe `Kyber` que definiremos na próxima seção, implementamos a codificação e decodificação de mensagens da seguinte forma.

```
1     def encode_message(self, message):
2         return vector(self.ntt_ring.field,
3                       [ceil(bit * self.params.q/2) for bit in message])
4
5     def decode_message(self, noisy_message):
6         def decode_coeff(c):
7             if abs(int(mod_centered(c, self.params.q))) <= self.params.q//4:
8                 return 0
9             return 1
10        return vector(self.ntt_ring.field, [decode_coeff(c) for c in noisy_message])
```

Pela hipótese da intratabilidade do MLWE decisional, $\mathbf{t}$ é indistinguível de um vetor aleatório de R_q^k mesmo para quem conhece $\mathbf{A}$. Então podemos usar $\mathbf{t}$ para gerar uma instância difícil do MLWE e esconder o polinômio $\text{Encode}(\mathbf{m})$ da seguinte forma. Escolha $\mathbf{r} \in R_q^k$ e $e_2 \in R_q$ segundo as distribuições $\mathcal{B}\left(R_q^k\right)$ e $\mathcal{B}\left(R_q\right)$, respectivamente. Como o polinômio $z = \langle \mathbf{t}, \mathbf{r} \rangle + e_2$ é uma instância do MLWE com segredo $\mathbf{r}$, então z é indistinguível de um polinômio aleatório de R_q. Portanto z pode ser usado para esconder a mensagem codificada obtendo $v = \text{Encode}(\mathbf{m}) + z$.

Note que, pela forma como $\mathbf{t}$ é definido, o polinômio v pode ser escrito como

$$\begin{aligned} v = \text{Encode}(\mathbf{m}) + z &= \text{Encode}(\mathbf{m}) + \langle \mathbf{t}, \mathbf{r} \rangle + e_2 \\ &= \text{Encode}(\mathbf{m}) + \langle \mathbf{A}\mathbf{s} + \mathbf{e}, \mathbf{r} \rangle + e_2 \\ &= \text{Encode}(\mathbf{m}) + \langle \mathbf{A}\mathbf{s}, \mathbf{r} \rangle + \langle \mathbf{e}, \mathbf{r} \rangle + e_2 \\ &= \text{Encode}(\mathbf{m}) + \left\langle \mathbf{s}, \mathbf{A}^\top \mathbf{r} \right\rangle + \langle \mathbf{e}, \mathbf{r} \rangle + e_2. \end{aligned}$$

O polinômio $\langle \mathbf{e}, \mathbf{r} \rangle + e_2$ tem coeficientes pequenos em relação a q, portanto a principal dificuldade de recuperar $\mathbf{m}$ a partir de v vem dos coeficientes grandes relativos à parcela $\left\langle \mathbf{s}, \mathbf{A}^\top \mathbf{r} \right\rangle$. Queremos então enviar uma dica, digamos $\mathbf{u}$, que permita ao destinatário remover a parcela densa da encriptação, e recuperar a mensagem $\mathbf{m}$. O principal desafio é que a dica $\mathbf{u}$ só deve ser útil para aquele que souber a chave secreta $\mathbf{s}$. Intuitivamente, uma boa solução seria um vetor $\mathbf{u}$ tal que $\langle \mathbf{s}, \mathbf{u} \rangle \approx \left\langle \mathbf{s}, \mathbf{A}^\top \mathbf{r} \right\rangle$.

Uma solução possível é tomar $\mathbf{u} = \mathbf{A}^\top \mathbf{r} + \mathbf{e}_1$, onde $\mathbf{e}_1$ é selecionado aleatoriamente segundo a distribuição $\mathcal{B}\left(R_q^k\right)$. Dessa forma, sob a intratabilidade do MLWE, o vetor $\mathbf{u}$

não revela informação sobre $\mathbf{r}$, e temos que

$$\langle \mathbf{s}, \mathbf{u} \rangle = \left\langle \mathbf{s}, \mathbf{A}^\top \mathbf{r} \right\rangle + \langle \mathbf{s}, \mathbf{e}_1 \rangle ,$$

onde $\langle \mathbf{s}, \mathbf{e}_1 \rangle$ é um vetor de polinômios de coeficientes pequenos.

Temos então que o par $(\mathbf{u}, z)$ contém $k+1$ amostras do MLWE relativo ao segredo $\mathbf{r}$. Supondo a intratabilidade do MLWE com parâmetros $(m, k+1, \mathcal{B}_\eta)$, esses valores são indistinguíveis de elementos aleatórios de $R_q^k \times R_q$. Portanto o par $(\mathbf{u}, v)$ encripta a mensagem $\mathbf{m}$.

Decriptação: Para a decriptação do texto encriptado $(\mathbf{u}, v)$, combinamos a chave secreta $\mathbf{s}$ com $\mathbf{u}$ e subtraímos o polinômio resultante de resultado de v, obtendo

$$\hat{c} = v - \langle \mathbf{s}, \mathbf{u} \rangle = \text{Encode}(\mathbf{m}) + \langle \mathbf{e}, \mathbf{r} \rangle - \langle \mathbf{s}, \mathbf{e}_1 \rangle + e_2.$$

Note como $\hat{c}$ é a mensagem codificada somada a um polinômio de erro dado por $\langle \mathbf{e}, \mathbf{r} \rangle - \langle \mathbf{s}, \mathbf{e}_1 \rangle + e_2$.

Para garantir que a decodificação seja possível sem erros, é preciso escolher os parâmetros q e η de forma que os coeficientes do polinômio de erro sejam pequenos em relação a q, porém tomando cuidado para manter o MLWE intratável.

1.5.4. Implementação do Kyber

O Kyber [Avanzi et al. 2019] consiste numa instanciação eficiente do esquema definido acima, com algumas otimizações. Em alto nível, o Kyber propõe 3 otimizações sobre o esquema protótipo que apresentamos

1. Definição de um anel polinomial R_q em que a NTT pode ser computada rapidamente, o que permite a multiplicação eficiente de polinômios de R_q.
2. A matriz pública $\mathbf{A} \in R_q^{k \times k}$ é representada por uma semente. Isso faz com que não sejam necessários $nk^2 \log_2(q)$ bits para representar a matriz. Além disso, a matriz $\mathbf{A}$ é calculada diretamente no domínio da NTT, já que ela só é usada para multiplicações.
3. O texto cifrado é comprimido $(\mathbf{u}, v)$ com perda, porém mantendo a probabilidade de falha de decriptação em patamares seguros.

Seleção de parâmetros e inicialização: Para cada nível de segurança, o Kyber usa um conjunto de parâmetros $(q, n, k, \eta_1, \eta_2, d_\mathbf{u}, d_v)$ diferentes. Já sabemos que o primo $q = 3229$ e $n = 256$ são fixos para todos os níveis. O parâmetro k representa a dimensão do módulo R_q^k usado. Os parâmetros η_1 e η_2 correspondem aos parâmetros de dispersão das binomiais centradas usadas para gerar instâncias do MLWE. Os parâmetros $d_\mathbf{u}$ e d_v podem ser vistos como índices de compressão dos elementos $\mathbf{u}$ e v que compõem o texto encriptado.

A tabela de parâmetros e a inicialização do Kyber são mostradas no código abaixo.

©2023 Sociedade Brasileira de Computação

```
@dataclass
class KyberParameters:
    q: int; n: int; k: int; eta1: int; eta2: int; du: int; dv: int;

class KyberPKE_CPA():
    SecurityParameters = {
        1: KyberParameters(q=3329, n=256, k=2, eta1=3, eta2=2, du=10, dv=4),
        3: KyberParameters(q=3329, n=256, k=3, eta1=2, eta2=2, du=10, dv=4),
        5: KyberParameters(q=3329, n=256, k=4, eta1=2, eta2=2, du=11, dv=5),
    }

    def __init__(self, security_level):
        self.params = self.SecurityParameters[security_level]
        self.ntt_ring = KyberNTTRing()
```

Geração de chaves: A geração de chaves é essencialmente igual à do protótipo mostrado na seção anterior. Primeiro escolha aleatoriamente a matriz $\mathbf{A}$ de $R_q^{k\times k}$, e vetores secretos $\mathbf{s}$ e $\mathbf{r}$ de acordo com $\mathcal{B}_{\eta_1}\left(R_q^k\right)$. Compute $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{t}$. As chaves pública e secreta são $\mathbf{pk} = (\mathbf{A}, \mathbf{t})$ e $\mathbf{sk} = (\mathbf{s})$, respectivamente.

A primeira diferença é que toda a aleatoriedade da geração de chaves parte de uma semente d de 256 bits selecionada aleatoriamente. Com base nessa semente e com o uso de XOFs e hashes criptográficos, todas as amostragens são feitas de forma determinística. A vantagem é que, para recriar o par de chaves, basta armazenar a semente d.

Há porém, diferenças mais importantes, principalmente em relação à representação da matriz $\mathbf{A}$ no domínio da NTT. Note que a matriz $\mathbf{A}$, tanto na geração de chaves quanto na encriptação, só é usada na multiplicação. Podemos então gerá-la diretamente no domínio da NTT, e como a NTT é uma função bijetora, gerar uma matriz $\hat{\mathbf{A}}$ uniforme no domínio da NTT é equivalente a gerar uma matriz $\mathbf{A}$ uniformemente de $R_q^{k\times k}$. Outra decisão importante é representar a matriz $\mathbf{A}$ através de uma semente pública de 256 bits, o que diminui a representação da chave pública ao custo da recomputação de $\mathbf{A}$ durante a encriptação.

Antes de definir a função de geração de chaves, considere as funções que fazem a amostragem da binomial centrada em vetores e polinômios, que denotamos por $\mathcal{B}\left(R_q^k\right)$ e $\mathcal{B}\left(R_q\right)$, respectivamente.

```
    def sample_polynomial_from_cbd(self, eta, xof):
        coeffs = [xof_centered_binomial_sample(xof, eta) for i in range(self.params.n)]
        return vector(self.ntt_ring.field, coeffs)

    def sample_vector_from_cbd(self, eta, xof):
        v = [self.sample_polynomial_from_cbd(eta, xof) for i in range(self.params.k)]
        return self.to_poly_vec(v)
```

Então a geração de chaves do Kyber pode ser implementada da seguinte forma. Note como toda a aleatoriedade da função de fato vem de uma variável d que é inicializada com 256 bits realmente aleatórios. Um hash G de 512 bits é aplicado sobre d de forma a obter dois valores de 256 bits, chamados ρ e σ. A variável ρ pode ser vista como a semente pública, que será usada para gerar $\mathbf{A}$, enquanto σ é semente secreta, usada para gerar os vetores secretos $\mathbf{s}$ e $\mathbf{e}$.

```
    def keygen(self):
        d = get_random_bytes(32)
        hash_g_of_d = HASH_G.new(d).digest()
```

©2023 Sociedade Brasileira de Computação

```
4          rho, sigma = hash_g_of_d[:16], hash_g_of_d[16:]
5          ntt_A = self.get_ntt_A_from_seed(rho)
6
7          sigma_xof = PRF.new(sigma)
8          s = self.sample_vector_from_cbd(self.params.eta1, sigma_xof)
9          e = self.sample_vector_from_cbd(self.params.eta1, sigma_xof)
10         ntt_s = s.ntt()
11         ntt_e = e.ntt()
12
13         ntt_t = self.matrix_poly_vec_product(ntt_A, ntt_s) + ntt_e
14         return (ntt_t, rho), (ntt_s)
```

Note como é necessário computar a NTT de $\mathbf{s}$ e $\mathbf{e}$ para poder fazer o produto e obter o vetor $\mathbf{t}$ no domínio da NTT. Note também como a função devolve o vetor público $\mathbf{t}$ e a chave secreta $\mathbf{s}$ no domínio da NTT, pois, durante a encriptação e decriptação, eles são usados somente em multiplicações.

Encriptação: Considere a função de encriptação em Sage abaixo.

```
1      def encrypt(self, pk, message, randomness):
2          (ntt_t, rho) = pk
3
4          ntt_A_transpose = self.get_ntt_A_from_seed(rho, transpose=True)
5
6          rand_vectors_xof = PRF.new(randomness)
7          r = self.sample_vector_from_cbd(self.params.eta1, rand_vectors_xof)
8          e1 = self.sample_vector_from_cbd(self.params.eta2, rand_vectors_xof)
9          e2  = self.sample_polynomial_from_cbd(self.params.eta2, rand_vectors_xof)
10         ntt_r = r.ntt()
11         u = self.matrix_poly_vec_product(ntt_A_transpose, ntt_r).inv_ntt() + e1
12
13         encoded_message = self.encode_message(message)
14         v = encoded_message + self.ntt_ring.inv_ntt(ntt_t * ntt_r) + e2
15
16         u_compressed = [self.compress(u_i, self.params.du) for u_i in u]
17         v_compressed = self.compress(v, self.params.dv)
18         return (u_compressed, v_compressed)
```

Diferente do algoritmo de encriptação apresentado no protótipo, a encriptação do Kyber deve ser determinística, com a fonte da aleatoriedade usada na função sendo derivada da semente passada como argumento pela variável `randomness`.

A função começa destrinchando a chave pública em $\hat{\mathbf{t}}$ e ρ, e computando a matriz $\hat{\mathbf{A}}$ através de ρ. Depois, são gerados $\mathbf{r}, \mathbf{e}_1$ e e_2 segundo as distribuições $\mathcal{B}_{\eta_1}\left(R_q^k\right)$, $\mathcal{B}_{\eta_2}\left(R_q^k\right)$ e $\mathcal{B}_{\eta_2}\left(R_q\right)$, respectivamente. Em seguida $\mathbf{u}$ e v são computados da mesma forma, porém no domínio da NTT, isto é

$$\begin{aligned}\mathbf{u} &= \mathbf{NTT}^{-1}\left(\mathbf{NTT}\left(\mathbf{A}\right)\cdot\mathbf{NTT}\left(\mathbf{r}\right)\right)+\mathbf{e}_1,\\ v &= \mathrm{Encode}\left(\mathbf{m}\right)+\mathbf{NTT}^{-1}\left(\left\langle\mathbf{NTT}\left(\mathbf{t}\right),\mathbf{NTT}\left(\mathbf{r}\right)\right\rangle\right)+e_2.\end{aligned}$$

A diferença crítica está na compressão de $\mathbf{u}$ e v. Após a compressão com perda de informação, cada coeficiente de cada polinômio de $\mathbf{u}$ passa a ser representado por $d_{\mathbf{u}}$ bits, enquanto cada coeficiente do polinômio v passa a ser representado por d_v bits.

Formalmente, a compressão e decompressão de um coeficiente c são dadas por

$$\mathbf{Compress}_d(c) = \left\lceil \frac{c2^d}{q} \right\rfloor \bmod 2^d,$$
$$\mathbf{Decompress}_d(\hat{c}) = \left\lceil \frac{\hat{c}q}{2^d} \right\rfloor.$$

Implementamos as funções de compressão e decompressão de polinômios da seguinte forma.

```
1   def compress(self, coefficients, d):
2       def compress_coefficient(x, d):
3           return mod(round(2**d / self.params.q * int(x)), 2**d)
4       return [compress_coefficient(c, d) for c in coefficients]
5
6   def decompress(self, coefficients, d):
7       def decompress_coefficient(x, d):
8           return round(self.params.q / 2**d * int(x))
9       return vector(self.ntt_ring.field,
10                    [decompress_coefficient(c, d) for c in coefficients])
```

Note que, como a compressão é com perda, após a decompressão pode haver um erro cujo tamanho depende do fator de compressão usado. Então, embora a compressão seja muito útil para gerar textos encriptados menores, é importante balancear os fatores de compressão para manter baixa a probabilidade de falha de decriptação.

Decriptação: A decriptação também é muito similar à decriptação mostrada no protótipo, exceto pela decompressão do texto encriptado feita logo no início da função.

```
1   def decrypt(self, sk, ciphertext):
2       ntt_s = sk
3
4       u_compressed, v_compressed = ciphertext
5       u = self.to_poly_vec([self.decompress(u_i, self.params.du)
6                             for u_i in u_compressed])
7       v = self.decompress(v_compressed, self.params.dv)
8
9       ntt_u = u.ntt()
10      noisy_message = v - self.ntt_ring.inv_ntt(ntt_s * ntt_u)
11
12      return self.decode_message(noisy_message)
```

1.6. Crystals-Dilithium

O Dilithium [Bai et al. 2021] é um esquema de assinatura cuja segurança é baseada na dificuldade dos problemas *learning with errors* (LWE) e *short integer solution* (SIS) em módulos, denotados por MLWE e MSIS, respectivamente. O esquema é construído segundo a heurística Fiat-Shamir, em que uma prova de conhecimento interativa é transformada num esquema de assinatura.

Nesta seção, apresentamos o Dilithium de forma construtiva da seguinte forma. Primeiro definimos o módulo-SIS (MSIS), que, juntamente com o MLWE já apresentado na seção anterior, é um problema em que se apoia a segurança do esquema. Em seguida, mostramos um protótipo de assinatura baseada em provas de conhecimento, explicando passo a passo como a transformação de Fiat-Shamir é aplicada. Após essa construção,

que forma a base do Dilithium, discutimos quais modificações são necessárias para a implementação eficiente. Por fim, apresentamos a implementação do Dilithium.

1.6.1. O anel polinomial usado pelo Dilithium

O Dilithium usa um mesmo anel polinomial $R_q = \mathbb{F}_q/(x^n+1)$, com $q = 8380417$ e $n = 256$, para todos os seus níveis de segurança. Neste caso, a raiz primitiva da unidade $\omega = 1753$ tem ordem 512 em $\mathbb{F}_q$, o que significa que podemos quebrar o polinômio $x^{256} + 1$ em 256 polinômios de grau 1 da forma

$$x^{256} + 1 = \prod_{i=0}^{255} \left(x - \omega^{2i+1}\right) = \prod_{i=0}^{255} \left(x - \omega^{2\texttt{bit_rev}(i,8)+1}\right).$$

De forma análoga ao Kyber, podemos implementar o anel R_q usado pelo Dilithium com o seguinte código.

```
class DilithiumNTTRing():

    def __init__(self):
        self.field = GF(8380417)
        self.n = 256
        self.omega_2n = self.field(1753)

    def ntt(self, a):
        return ntt_leveled_negacyclic_256(a, self.field, self.omega_2n, 8)

    def inv_ntt(self, a_hat):
        return inv_ntt_leveled_negacyclic_256(a_hat, self.field, self.omega_2n, 8)

    def poly_product_ntt_domain(self, a_hat, b_hat):
        return vector(self.field, [c_a * c_b for c_a, c_b in zip(a_hat, b_hat)])
```

Note como a NTT rápida pode ser calculada completamente em 8 níveis. Além disso, o produto de polinômios no domínio da NTT é mais simples do que aquele do Kyber, pois os resíduos são números, não polinômios de grau 1. Isso faz com que o produto seja simplesmente a multiplicação em $\mathbb{F}_q$.

1.6.2. Os problemas usados pelo Dilithium

A segurança do Dilithium é baseada em dois problemas: o MLWE e o MSIS. O MLWE é responsável por garantir a segurança da chave secreta, enquanto o MSIS garante a dificuldade em forjar assinaturas. Lembre que, na Seção 1.5.2, é introduzido o MLWE usado pelo Kyber. A variante do problema usada pelo Dilithium é muito similar, exceto pela distribuição de vetores pequenos, que no lugar da binomial centrada é a uniforme.

Definição 4 (Problema MLWE computacional para o Dilithium)**.** Considere o conjunto S_η de polinômios de R_q cuja norma infinito é menor ou igual a η. Formalmente

$$S_\eta = \left\{s \in R_q : \|s\|_\infty \leq \eta\right\}.$$

Sorteie uma matriz $\mathbf{A}$ aleatoriamente de forma uniforme de $R_q^{k\times\ell}$. Escolha dois vetores $\mathbf{s}_1$ e $\mathbf{s}_2$ aleatoriamente de forma uniforme dos módulos S_η^ℓ e S_η^k, respectivamente. Compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$. O problema MLWE computacional de parâmetros (m,k,η) consiste em, dados $\mathbf{A}$ e $\mathbf{t}$, encontrar o vetor $\mathbf{s}_1$.

□

O problema MSIS consiste em encontrar soluções pequenas de um sistema linear homogêneo em módulos. Este problema é definido formalmente abaixo.

Definição 5 (Problema MSIS computacional)**.** Seja $\mathbf{A}$ uma matriz aleatória tomada uniformemente de $R_q^{m \times k}$. O problema MSIS computacional na forma normal com parâmetros (m, k, γ) consiste em encontrar $\mathbf{y} \in R_q^{(m+k)}$ tal que $\|\mathbf{y}\|_\infty \le \gamma$ e $[\,\mathbf{I} \mid \mathbf{A}\,]\,\mathbf{y} = \mathbf{0}$. □

1.6.3. Assinatura baseada em provas de conhecimento

Suponha que sabemos a solução de uma instância do MLWE. Isto é, seja $\mathbf{A} \in R_q^{k \times \ell}$ uma matriz de polinômios e $\mathbf{t} \in R_q^{\ell}$, nós temos em mãos vetores pequenos $\mathbf{s}_1 \in R_q^{\ell}$ e $\mathbf{s}_2 \in R_q^{k}$ tais que

$$\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2 = \mathbf{t}.$$

Dado o par $(\mathbf{A}, \mathbf{t})$, uma prova de conhecimento de $(\mathbf{s}_1, \mathbf{s}_2)$ é uma construção criptográfica que satisfaz as seguintes propriedades aparentemente contraditórias:

1. convence um verificador que de fato conhecemos uma solução $(\mathbf{s}_1, \mathbf{s}_2)$;
2. não permite que o verificador aprenda nenhuma informação sobre $\mathbf{s}_1$.

A solução tipicamente usada para esse problema é usar uma prova interativa entre o provador e o verificador. A interação consiste em o verificador fazer perguntas que devem ser respondidas pelo provador de forma satisfatória.

Nesta seção, vamos mostrar como construir uma prova de conhecimento de uma solução do MLWE. Depois, mostramos como esta construção pode ser transformada num esquema de assinatura digital por meio da transformada de Fiat-Shamir.

Conjuntos e funçõs auxiliares: Na prova interativa, vamos precisar de dois conjuntos de polinômios pequenos definidos a seguir.

Dado um inteiro $\gamma \ge 0$, definimos o conjunto S_γ como o conjunto de polinômios em R_q de norma infinito no máximo γ. Formalmente

$$S_\gamma = \left\{ w \in R_q : \|w\|_\infty \le \gamma \right\}.$$

Para um inteiro $\tau \ge 0$, o conjunto $B_\tau \subset R_q$ contém os polinômios com coeficientes em $\{-1, 0, 1\}$ mas com apenas τ coeficientes diferentes de 0. Isto é

$$B_\tau = \left\{ w = w_0 + \ldots + w_{n-1}x^{n-1} \in R_q : -1 \le w_i \le 1 \text{ e } \sum_{i=0}^{n-1} |w_i| = \tau \right\}.$$

Protótipo de prova de conhecimento: A Figura 1.4 mostra o protótipo da prova de conhecimento. Note que, embora útil para entender a construção do esquema, este protótipo é inseguro, como veremos adiante. Neste protocolo, o provador tenta convencer o

©2023 Sociedade Brasileira de Computação

verificador de que ele sabe uma solução $(\mathbf{s}_1, \mathbf{s}_2)$ do MLWE tal que $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$, para uma matriz $\mathbf{A} \in R_q^{k\times\ell}$ e um vetor $\mathbf{t} \in R_q^{\ell}$.

O provador começa selecionando um vetor pequeno $\mathbf{y}$ chamado máscara, e computando $\mathbf{w} = \mathbf{A}\mathbf{y}$. Note que $\mathbf{w}$ é um vetor denso e é possível provar que, quando γ_1 é suficientemente grande, $\mathbf{w}$ tem distribuição indistinguível da uniforme sobre R_q^k, e portanto não revela nada sobre $\mathbf{y}$ [Peikert 2015, Seção 4.1.1]. O vetor $\mathbf{w}$ é então enviado ao verificador.

Ao receber $\mathbf{w}$, o verificador gera um polinômio uniformemente aleatório de B_τ, chamado desafio, e envia-o ao provador. O provador então computa $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ e envia-o ao verificador. O protocolo termina com o verificador realizando duas verificações. A primeira é se o vetor $\mathbf{z}$ recebido de fato é pequeno. A segunda é se o vetor $\mathbf{w}$ recebido no protocolo próximo de $\mathbf{A}\mathbf{z} - c\mathbf{t}$. Se as duas verificações passarem, o verificador aceita a prova. Caso contrário, a prova é rejeitada.

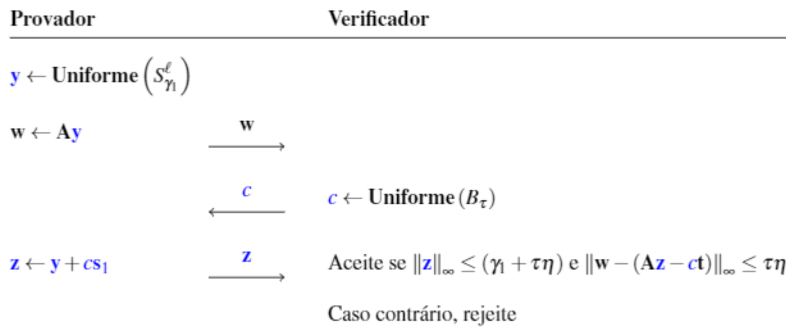

Figura 1.4. Protótipo inseguro da prova de conhecimento interativa ainda pois a resposta final vaza informação sobre o segredo.

Primeiro, vejamos que alguém honesto e que sabe o segredo $\mathbf{s}_1$ consegue passar na prova. Neste caso, o provador computou honestamente $\mathbf{w} = \mathbf{A}\mathbf{y}$ e $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$. Note que, como a soma dos valores absolutos dos coeficientes de c é τ, vale que $\|c\mathbf{s}_1\|_\infty \leq \tau\eta$. Então, para a primeira verificação temos que

$$\|\mathbf{z}\|_\infty = \|\mathbf{y} + c\mathbf{s}_1\|_\infty \leq \|\mathbf{y}\|_\infty + \|c\mathbf{s}_1\|_\infty \leq \gamma_1 + \tau\eta.$$

Para a segunda verificação, observamos que

$$\begin{aligned}\mathbf{A}\mathbf{z} - c\mathbf{t} &= \mathbf{A}(\mathbf{y} + c\mathbf{s}_1) - c\mathbf{t} \\ &= \mathbf{A}(\mathbf{y}) + \mathbf{A}(c\mathbf{s}_1) - c\mathbf{t} \\ &= \mathbf{w} + c(\mathbf{t} - \mathbf{s}_2) - c\mathbf{t} \\ &= \mathbf{w} - c\mathbf{s}_2.\end{aligned}$$

Portanto, vale que $\mathbf{w} - (\mathbf{A}\mathbf{z} - c\mathbf{t}) = c\mathbf{s}_2$, que é um vetor de norma $\|c\mathbf{s}_2\|_\infty \leq \tau\eta$.

Considere o desempenho de um falso provador que não sabe $(\mathbf{s}_1, \mathbf{s}_2)$. Suponha que, após ele enviar a mensagem inicial $\mathbf{w}$, há somente um desafio c que ele sabe responder corretamente. Isso pode ocorre, por exemplo, quando o falso provador observou uma

©2023 Sociedade Brasileira de Computação

interação passada do provador real. Para proteger o esquema contra esse tipo de ataque, basta que o conjunto B_τ seja grande o suficiente para que probabilidade de um mesmo desafio ser gerado duas vezes seja desprezável.

Suponha então que o falso provador, após enviar $\mathbf{w}$, é capaz de responder pelo menos 2 desafios diferentes c_1 e c_2, com respostas válidas $\mathbf{z}_1$ e $\mathbf{z}_2$. Neste caso, devido à validade das respostas, temos que

$$\mathbf{w} = \mathbf{A}\mathbf{z}_1 - c_1\mathbf{t} + \mathbf{e}_1 = \mathbf{A}\mathbf{z}_2 - c_2\mathbf{t} + \mathbf{e}_2,$$

onde $\mathbf{e}_1$ e $\mathbf{e}_2$ são vetores pequenos de norma infinito menor ou igual a $\tau\eta$. Portanto, vale que

$$\mathbf{A}(\mathbf{z}_1 - \mathbf{z}_2) + (c_2 - c_1)\mathbf{t} + (\mathbf{e}_1 - \mathbf{e}_2) = \mathbf{0}$$
$$\left[\,\mathbf{I} \mid \mathbf{A} \mid \mathbf{t}\,\right] \begin{bmatrix} \mathbf{e}_1 - \mathbf{e}_2 \\ \hline (\mathbf{z}_1 - \mathbf{z}_2)^\top \\ \hline c_1 - c_2 \end{bmatrix} = \mathbf{0},$$

o que implica que o falso provador consegue resolver o MSIS para a matriz $[\,\mathbf{I} \mid \mathbf{A} \mid \mathbf{t}\,]$.

Embora a dificuldade em dar uma resposta válida seja garantida pela dificuldade do MSIS, e a dificuldade em quebrar a chave secreta a partir da pública seja protegida pelo MLWE, há ainda dois problemas muito sérios no protocolo apresentado. O primeiro é que o verificador consegue computar $c\mathbf{s}_2 = \mathbf{w} - (\mathbf{A}\mathbf{z} - c\mathbf{t})$, e portanto pode obter o vetor secreto $\mathbf{s}_2$ com uma simples divisão de polinômio, e usá-lo para obter $\mathbf{s}_1$.

O segundo problema é que o vetor $\mathbf{z}$ é muito dependente do vetor secreto $\mathbf{s}_1$ pois a máscara $\mathbf{y}$ usada para ofuscá-lo é um polinômio pequeno. Então, observando um número suficientemente grande de interações, um atacante poderia ser capaz de obter alguma informação sobre $\mathbf{s}_1$.

Protegendo $\mathbf{s}_1$ com amostragem por rejeição: Queremos garantir que o verificador, que conhece c e $\mathbf{z}$, não consiga obter nenhuma informação sobre $\mathbf{s}_1$ e $\mathbf{s}_2$. Primeiramente, considere o caso de $\mathbf{s}_1$. Formalmente, queremos que $\Pr(\mathbf{s}_1)$ seja independente de $(c, \mathbf{z})$, isto é, que $\Pr(\mathbf{s}_1 | c, \mathbf{z}) = \Pr(\mathbf{s}_1)$.

Fixe um segredo $\mathbf{s}_1$, e considere a distribuição de probabilidade conjunta $\Pr(c, \mathbf{z})$

$$\begin{aligned} \Pr(c, \mathbf{z}) &= \Pr(\mathbf{z} \mid c) \Pr(c) \\ &= \Pr_{\mathbf{y}}(\mathbf{z} = \mathbf{y} + c\mathbf{s}_1 \mid c) \Pr(c) \\ &= \Pr_{\mathbf{y}}(\mathbf{y} = \mathbf{z} - c\mathbf{s}_1 \mid c) \Pr(c). \end{aligned}$$

Como $\mathbf{y}$ é um vetor uniformemente aleatório de $S^\ell_{\gamma_1}$, então se a condição $(\mathbf{z} - c\mathbf{s}_1) \in S^\ell_{\gamma_1}$ fosse satisfeita, valeria que

$$\Pr_{\mathbf{y}}(\mathbf{y} = \mathbf{z} - c\mathbf{s}_1 \mid c) = \frac{1}{\left|S^\ell_{\gamma_1}\right|},$$

que caracteriza uma distribuição uniforme sobre S_{γ_1}. Logo, se o provador abortar e recomeçar o protocolo sempre que $(\mathbf{z} - c\mathbf{s}_1) \notin S_{\gamma_1}^{\ell}$, então os valores de $(\mathbf{z}, c)$ serão de fato independentes de $\mathbf{s}_1$.

Podemos simplificar a condição de rejeição da seguinte forma. Como $\mathbf{s}_1 \in S_{\eta}^{\ell}$ e $c \in B_{\tau}$, então $\|c\mathbf{s}_1\|_{\infty} \leq \eta\tau$. Considerando então a desigualdade triangular sobre $(\mathbf{z} - c\mathbf{s}_1)$, obtemos

$$\|\mathbf{z} - c\mathbf{s}_1\|_{\infty} \leq \|\mathbf{z}\|_{\infty} + \|c\mathbf{s}_1\|_{\infty} \leq \|\mathbf{z}\|_{\infty} + \eta\tau.$$

Podemos então garantir que $\|\mathbf{z} - c\mathbf{s}_1\|_{\infty} \leq \gamma_1$ se forçarmos que o lado direito da inequação acima satisfaça

$$\|\mathbf{z}\|_{\infty} + \eta\tau \leq \gamma_1.$$

Ou seja, se aceitarmos $\|\mathbf{z}\|_{\infty}$ somente quando $\|\mathbf{z}\|_{\infty} \leq \gamma_1 - \eta\tau$, garantimos que $\|\mathbf{z} - c\mathbf{s}_1\|_{\infty} \leq \gamma_1$, e, portanto $\mathbf{z}$ pode ser enviado ao verificador sem vazamento de informação sobre $\mathbf{s}_1$.

Protegendo $\mathbf{s}_2$ com amostragem por rejeição: Considere agora o caso de $\mathbf{s}_2$, cujos coeficientes são vazados pois o verificador conhece todos os termos do lado direito da equação.

$$c\mathbf{s}_2 = \mathbf{w} - (\mathbf{A}\mathbf{z} - c\mathbf{t}).$$

Note que, como o vetor $c\mathbf{s}_2$ revelado tem coeficientes pequenos, a fonte de vazamento de informação está nos bits menos significativos dos coeficientes dos polinômios de $\mathbf{w}$. Intuitivamente, se o provador enviasse somente os bits mais significativos de $\mathbf{w}$, os bits menos significativos de $\mathbf{w}$ tratariam de esconder perfeitamente o vetor $c\mathbf{s}_2$.

Para formalizar esta ideia, vamos definir a função que, dado um parâmetro de α, decompõe um elemento r de $\mathbb{F}_q$ em duas partes r_1 e r_0 tais que

1. $r = r_0 + \alpha r_1$ e
2. $-(\alpha/2) \leq r_0 < (\alpha/2)$.

```
1    def decompose(self, r, alpha):
2        r = int(r) % self.params.q
3        r0 = mod_centered(r, alpha)
4        if r - r0 == self.params.q - 1:
5            return (0, r0 - 1)
6        return (r - r0) // alpha, r0
```

Usando a função decompose, podemos implementar funções específicas que devolvem a parte mais significativa r_1 ou a menos r_0.

```
1    def high_bits_coefficient(self, r, alpha):
2        return self.decompose(r, alpha)[0]
3
4    def low_bits_coefficient(self, r, alpha):
5        return self.decompose(r, alpha)[1]
```

Também usaremos a extensão das funções acima para vetores de polinômios, chamadas HighBits e LowBits, que simplesmente aplicam as respectivas funções a cada um dos coeficientes de cada polinômio no vetor.

```
def high_bits(self, r, alpha):
    v = [[self.high_bits_coefficient(c, alpha) for c in poly] for poly in r]
    return self.to_poly_vec(v)

def low_bits(self, r, alpha):
    v = [[self.low_bits_coefficient(c, alpha) for c in poly] for poly in r]
    return self.to_poly_vec(v)
```

Suponha então que provador deixe de enviar $\mathbf{w}$ e envie $\mathbf{w}_1 = \text{HighBits}(\mathbf{w}, 2\gamma_2)$ em seu lugar. Considere o valor computado para a verificação

$$\text{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2) = \text{HighBits}(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2).$$

Podemos ver que ainda é possível que alguma informação sobre $c\mathbf{s}_2$ vaze para $\text{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$ por meio dos carregamentos que ocorrem durante a computação de

$$\text{LowBits}(\mathbf{w}, 2\gamma_2) - \text{LowBits}(c\mathbf{s}_2, 2\gamma_2) = \text{LowBits}(\mathbf{w}, 2\gamma_2) - c\mathbf{s}_2.$$

Considere $\mathbf{w}_0 = \text{LowBits}(\mathbf{w}, 2\gamma_2)$. Queremos então garantir que $\Pr(\mathbf{s}_2)$ seja independente de $\Pr(c, \mathbf{w}_0)$, isto é, que $\Pr(\mathbf{s}_2 | c, \mathbf{w}_0) = \Pr(\mathbf{s}_2)$.

Mas isso é exatamente análogo ao que computamos na seção anterior para $\mathbf{s}_1$ e $\mathbf{z}$, com os vetores $\mathbf{s}_2$ e $\mathbf{w}_0$ em seus lugares. Note que, pela definição de LowBits, qualquer vetor em $S_{\gamma_2 - 1}$ é um vetor $\|\mathbf{w}_0\|_\infty$ possível[3]. Portanto, usando o resultado mostrado na seção anterior, podemos garantir que

$$\Pr(\mathbf{s}_2 | c, \mathbf{w}_0) = \Pr(\mathbf{s}_2),$$

se somente aceitarmos valores de $\mathbf{w}_0$ tais que $\|\mathbf{w}_0\|_\infty < \gamma_2 - \eta\tau$. Esta condição pode ser reescrita como

$$\text{LowBits}(\mathbf{Ay} - c\mathbf{s}_2, 2\gamma_2) < \gamma_2 - \eta\tau.$$

Então, caso a condição não seja satisfeita, o provador aborda e recomeça o protocolo com um novo $\mathbf{y}$.

Construindo um esquema de assinatura: Uma assinatura digital pode ser vista como uma demonstração não interativa de que conhecemos ao mesmo tempo a chave secreta e a mensagem a ser assinada. Então, podemos transformar a prova de conhecimento acima num esquema de assinatura se fizermos o desafio $\mathbf{c}$ depender da mensagem e removermos a interação.

Suponha que há uma função de hash H que devolve polinômios de B_τ. A transformação de Fiat-Shamir consiste em eliminar o verificador e computar o desafio $\mathbf{c}$ como $\mathbf{c} = H(\mathbf{w}, \mathbf{m})$, onde $\mathbf{m}$ é a mensagem a ser assinada. Sendo H uma função de hash criptográfica segura, isso garante que seja computacionalmente inviável gerar $\mathbf{c}$ antes de $\mathbf{w}$ e reusar o mesmo par $(\mathbf{c}, \mathbf{w})$ para uma mensagem $\mathbf{m}$ diferente.

Com esta transformação, e adicionando a repetição do protocolo para aceitar vetores $\|\mathbf{z}\|_\infty$ somente quando $\|\mathbf{z}\|_\infty < \gamma_1$, obtemos o esquema de assinatura cuja segurança é baseada na dificuldade dos problemas MLWE e MSIS abaixo.

[3]Note que usamos $\gamma_2 - 1$ pois a $\text{LowBits}(w, 2\gamma_2)$ devolve um coeficiente w_0 no intervalo $-\gamma_2 \leq w_0 < \gamma_2$.

©2023 Sociedade Brasileira de Computação

Geração de chaves	Assinar $(\mathbf{sk} = (\mathbf{s}_1, \mathbf{s}_2), \mathbf{m})$	Verificar $(\mathbf{pk} = (\mathbf{A}, \mathbf{t}), (c, \mathbf{z}))$
$\mathbf{A} \leftarrow \textbf{Uniforme}\left(R_q^{k\times\ell}\right)$ $\mathbf{s}_1 \leftarrow \textbf{Uniforme}\left(S_\eta^k\right) \in R_q^k$ $\mathbf{s}_2 \leftarrow \textbf{Uniforme}\left(S_\eta^\ell\right) \in R_q^\ell$ $\mathbf{t} \leftarrow \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ $\mathbf{pk} \leftarrow (\mathbf{A}, \mathbf{t})$ $\mathbf{sk} \leftarrow (\mathbf{s}_1, \mathbf{s}_2)$ Devolva a assinatura$(c, \mathbf{z})$	**Faça** : $\mathbf{y} \leftarrow \textbf{Uniforme}\left(S_{\gamma_1}^\ell\right)$ $\mathbf{w} \leftarrow \mathbf{A}\mathbf{y}$ $\mathbf{w}_1 \leftarrow \text{HighBits}(\mathbf{w}, 2\gamma_2)$ $c \leftarrow \text{Hash}(\mathbf{w}_1, \mathbf{m})$ $\mathbf{z} \leftarrow \mathbf{y} + c\mathbf{s}_1$ **Enquanto** 1. $\|\mathbf{z}\|_\infty > \gamma_1 - \eta\tau$, ou 2. $\text{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2) \geq \gamma_2 - \eta\tau$ **Devolva** a assinatura$(c, \mathbf{z})$	$\mathbf{w}_1 \leftarrow \text{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$ $c' = \text{Hash}(\mathbf{w}_1, \mathbf{m})$ Aceite se $c' = c$ e $\|\mathbf{z}\|_\infty \leq \gamma_1$

Figura 1.5. Esquema de assinatura usando a transformação de Fiat-Shamir com amostragem por rejeição.

Esse esquema de assinatura é a base do Dilithium. As diferenças principais estão nas várias otimizações para comprimir a chave pública e as assinaturas, além do uso de um anel R_q que admite a computação rápida da NTT completa para a multiplicação de polinômios.

1.6.4. Implementação do Dilithium

O Dilithium consiste na otimização do esquema mais simples apresentado na seção anterior. As otimizações são análogas às usadas no Kyber, como a representação de $\mathbf{A}$ pela semente, o uso da NTT para as operações de multiplicação entre polinômios e compressão das chaves públicas.

Seleção de parâmetros e inicialização: Por conta das várias otimizações que são aplicadas, o Dilithium é usa um grande número de parâmetros para definir cada nível de segurança. Os parâmetros $n = 256$ e $q = 8380417$ que definem o anel R_q são fixos para todos os níveis de segurança. Os parâmetros k e ℓ definem o espaço da matriz pública $\mathbf{A} \in R_q^{k\times\ell}$, enquanto η define o conjunto S_η^ℓ de onde é tirada a chave secreta $\mathbf{s}_1$.

O parâmetro τ define o raio do conjunto B_τ de onde os desafios $\mathbf{c}$ são tirados. O parâmetro d é um índice de compressão da chave pública. O inteiro γ_1 define o conjunto S_{γ_1} de máscaras $\mathbf{y}$, enquanto γ_2 é usada para definir as funções HighBits e LowBits. Finalmente, $\beta = \eta\tau$ e ω são números usados para definir os limites das normas de vetores que são esperados quando a assinatura é válida.

```
@dataclass
class DilithiumParameters:
    q: int; n: int; k: int; l: int; tau: int; eta: int; d: int;
    gamma1: int; gamma2: int; beta: int; omega: int

class Dilithium():
    SecurityParameters = {
        2: DilithiumParameters(
            q=8380417, d=13, tau=39, gamma1=2**17, gamma2=95232,
            n=256, k=4, l=4, eta=2, beta=78, omega=80),
        3: DilithiumParameters(
            q=8380417, d=13, tau=49, gamma1=2**19, gamma2=261888,
            n=256, k=6, l=5, eta=4, beta=196, omega=55),
```

```
14          5: DilithiumParameters(
15              q=8380417, d=13, tau=60, gamma1=2**19, gamma2=261888,
16              n=256, k=8, l=7, eta=2, beta=120, omega=75),
17      }
18
19      def __init__(self, security_level):
20          self.params = self.SecurityParameters[security_level]
21          self.ntt_ring = DilithiumNTTRing()
```

Geração de chaves: A geração de chaves do Dilithium é muito similar à mostrada na Figura 1.5, exceto pela compressão da chave pública. Considere o código da geração de chaves abaixo.

```
1   def keygen(self):
2       zeta = get_random_bytes(32)
3       xof_h = XOF_H.new(zeta)
4       rho = xof_h.read(32)
5       rho_prime = xof_h.read(64)
6
7       ntt_A = self.get_ntt_A_from_seed(rho)
8       s1, s2 = self.expand_S(rho_prime)
9       ntt_s1 = s1.ntt()
10
11      t = self.matrix_poly_vec_product(ntt_A, ntt_s1).inv_ntt() + s2
12      t1, t0 = self.power2round_poly_vector(t, self.params.d)
13      pk_hash = self.hash_H(rho + t1.as_bytes(), 32)
14
15      assert (t1 * 2**self.params.d == t - t0)
16      return (rho, t1), (rho, pk_hash, s1, s2, t0)
```

Podemos ver que, como no Kyber, a matriz $\mathbf{A}$ é representada compactamente pela semente e as multiplicações são todas feitas no domínio da NTT. Para a compressão da chave pública, o vetor $\mathbf{t}$ é dividido em dois vetores $\mathbf{t}_1$ e $\mathbf{t}_0$ tais que $\mathbf{t} = \mathbf{t}_1 2^d + \mathbf{t}_0$, sendo que apenas $\mathbf{t}_1$ faz parte da chave pública. A a parte menos significativa $\mathbf{t}_0$ é guardada na chave secreta, pois ela será importante para gerar assinaturas válidas. A função `power2round_poly_vector` que quebra $\mathbf{t}$ em $\mathbf{t}_1$ e $\mathbf{t}_0$ está implementada abaixo.

```
1   def power2round(self, r, d):
2       r = int(r) % self.params.q
3       r0 = mod_centered(r, 2**d)
4       return (r - r0)//2**d, r0
5
6   def power2round_poly_vector(self, poly_vector, d):
7       poly_vector0 = [[] for _ in poly_vector]
8       poly_vector1 = [[] for _ in poly_vector]
9       for i, poly in enumerate(poly_vector):
10          for r in poly:
11              r1, r0 = self.power2round(r, d)
12              poly_vector0[i].append(r0)
13              poly_vector1[i].append(r1)
14      return self.to_poly_vec(poly_vector1), self.to_poly_vec(poly_vector0)
```

Assinatura: Devido à compressão de chave pública usada pelo Dilithium, a assinatura é significativamente mais complicada do que aquela mostrada na Figura 1.5. A assinatura pode ser implementada da seguinte forma.

```
1   def sign(self, sk, message_bytes):
2       (rho, pk_hash, s1, s2, t0) = sk
3       ntt_A = self.get_ntt_A_from_seed(rho)
4       mu = self.hash_H(pk_hash + message_bytes, 64)
```

©2023 Sociedade Brasileira de Computação

```
5
6        sigma_prime = get_random_bytes(64)
7        s1_ntt, s2_ntt, t0_ntt = s1.ntt(), s2.ntt(), t0.ntt()
8
9        kappa, (z, h) = 0, (None, None)
10       while h == None:
11           y = self.expand_mask(sigma_prime, kappa.to_bytes(32))
12           kappa += self.params.l
13
14           w_c_z = self.get_w_c_z(y, ntt_A, s1_ntt, s2_ntt, mu)
15           if w_c_z:
16               (w, w1), (c_tilde, c_ntt), z = w_c_z
17               h = self.get_hints_for_w((w, w1), t0_ntt, s2_ntt, c_ntt)
18
19       return (c_tilde, z, h)
```

Em alto nível, o algoritmo de assinatura segue aquele do esquema apresentado na Figura 1.5. O núcleo da função de assinatura é o laço de rejeições, que garante que a assinatura não vaza informação sobre a chave secreta.

O laço começa da mesma forma que vimos na Figura 1.5, primeiro selecione $\mathbf{y}$ e tente gerar $(\mathbf{w}, c, \mathbf{z})$ válidos. Isto é, vetores fora dos dos intervalos de rejeição. A geração da máscara $\mathbf{y}$ é feita pela função expand_mask, que descrita abaixo. Note como é usado um XOF para gerar os inteiros no intervalo de $(-\gamma_1 + 1)$ a γ_1.

```
1    def expand_mask(self, rhoprime, kappa):
2        xof = SHAKE256.new(rhoprime + kappa)
3        y = []
4        min_inclusive, max_exclusive = -self.params.gamma1 + 1, self.params.gamma1 + 1
5        for i in range(self.params.l):
6            y.append([xof_randrange(xof, min_inclusive, max_exclusive)
7                      for j in range(self.params.n)])
8
9        return PolynomialVector(self.ntt_ring, y)
```

O procedimento que tenta gerar $(\mathbf{w}, c, \mathbf{z})$ válidos é implementado pela função get_w_c_z abaixo. Note como, ao final da função, as duas condições de rejeição são testadas. Caso qualquer das duas seja inválida, a função devolve None para indicar a falha.

```
1    def get_w_c_z(self, y, ntt_A, s1_ntt, s2_ntt, mu):
2        y_ntt = y.ntt()
3
4        w = self.matrix_poly_vec_product(ntt_A, y_ntt).inv_ntt()
5        w1 = self.high_bits(w, 2 * self.params.gamma2)
6
7        c_tilde = self.hash_H(mu + w1.as_bytes(), 32)
8        c = self.sample_in_ball(c_tilde)
9        c_ntt = self.ntt_ring.ntt(c)
10       z = y + (s1_ntt * c_ntt).inv_ntt()
11
12       r0 = self.low_bits(w - (s2_ntt * c_ntt).inv_ntt(), 2 * self.params.gamma2)
13       z_norm_cond = (z.norm_infinity() >= self.params.gamma1 - self.params.beta)
14       r0_norm_cond = (r0.norm_infinity() >= self.params.gamma2 - self.params.beta)
15
16       if z_norm_cond or r0_norm_cond:
17           return None
18
19       return ((w, w1), (c_tilde, c_ntt), z)
```

Exceto pelas multiplicações no domínio da NTT, a única diferença fundamental entre a função acima e o código dentro do laço na Figura 1.5 é em como o polinômio c é representado. Por motivos de eficiência, o Dilithium representa o polinômio $c \in B_\tau$

©2023 Sociedade Brasileira de Computação

através da semente $\tilde{c}$ usada para gerá-lo. Dada uma semente $\tilde{c}$, a função `sample_in_ball` instancia um XOF com o SHAKE256. Este XOF é então usado para gerar os τ coeficientes em $\{-1,1\}$ usando uma variação do algoritmo de Fisher-Yates.

```
def sample_in_ball(self, c_tilde):
    xof = SHAKE256.new(c_tilde)
    c = [0] * 256
    for i in range(256 - self.params.tau, 256):
        j = xof_randrange(xof, 0, i + 1)
        c[i] = c[j]
        c[j] = (-1)**xof_random_bit(xof)

    assert(c.count(1) + c.count(-1) == self.params.tau)
    return c
```

Após encontrados valores $(\mathbf{w}, c, \mathbf{z})$ válidos, entra em ação um procedimento particular do Dilithium que chamamos de `get_hints_for_w`. Este procedimento é responsável por produzir um vetor binário h chamado vetor de dicas, que é necessário para garantir a corretude da verificação usando a chave pública compactada $\mathbf{t}_1$.

Garantindo que a compressão de chaves não afeta a verificação: Primeiro, vejamos como a compressão de chaves dificulta o procedimento de verificação simplificado mostrado na Figura 1.5. Vimos que o primeiro passo da verificação consiste em computar $\mathbf{w}_1 = \text{HighBits}(\mathbf{Az} - c\mathbf{t}, 2\gamma_2)$.

Porém, a compressão de $\mathbf{t}$ implica que apenas a sua parte mais significativa $\mathbf{t}_1$ seja conhecida pelo verificador. Como ainda não é possível computar $\mathbf{w}_1$, o verificador pode computar o vetor

$$\hat{\mathbf{w}} = \mathbf{Az} - c2^d\mathbf{t}_1 = \mathbf{Az} - c\mathbf{t} + c\mathbf{t}_0 = \mathbf{w} - c\mathbf{s}_2 + c\mathbf{t}_0.$$

Formalmente, o problema enfrentado pelo verificador é que, em geral

$$\mathbf{w}_1 = \text{HighBits}(\mathbf{w} - c\mathbf{s}_2) \neq \text{HighBits}(\hat{\mathbf{w}}),$$

por conta do fator $c\mathbf{t}_0$. Uma solução possível seria incluir o vetor $\mathbf{t}_0$, que não precisa ser secreto, na assinatura. Assim, o verificador, poderia remover o fator $c\mathbf{t}_0$ e computar $\mathbf{w}_1$. Porém, isso aumentaria muito o tamanho das assinaturas, e não justificaria a vantagem obtida pela compressão da chave pública. A solução usada no Dilithium consiste em computar um vetor conciso de dicas que permitem que o verificador compute $\mathbf{w}_1$ corretamente.

Antes de mostrarmos como tal vetor de dicas é calculado, considere a seguinte propriedade das funções HighBits e LowBits. Fixe um inteiro r e sejam $r_1 = \text{HighBits}(r, 2\gamma_2)$ e $r_0 = \text{LowBits}(r, 2\gamma_2)$, de forma que $r = \gamma_2 r_1 + r_0$, e $\gamma_2 \leq r_0 < \gamma_2$. Suponha que $-\gamma_2 \leq a < \gamma_2$, então

$$\text{HighBits}(r+a) = \begin{cases} r_1, \text{ se } -\gamma_2 \leq r_0 + a < \gamma_2, \\ r_1 + 1, \text{ se } r_0 + a \geq \gamma_2, \\ r_1 - 1, \text{ se } r_0 + a < -\gamma_2. \end{cases}$$

Podemos interpretar a observação acima da seguinte forma. Se a é um inteiro desconhecido tal que $-\gamma_2 \leq a < \gamma_2$. Se soubermos que $\text{HighBits}(r+a, 2\gamma_2) \neq$

©2023 Sociedade Brasileira de Computação

$\text{HighBits}(r, 2\gamma_2)$, então necessariamente

$$\text{HighBits}(r+a, 2\gamma_2) = \begin{cases} \text{HighBits}(r, 2\gamma_2) + 1, \text{ se } r_0 \geq 0, \\ \text{HighBits}(r, 2\gamma_2) - 1, \text{ se } r_0 < 0. \end{cases}$$

De forma que, se soubermos r, apenas 1 bit nos permite descobrir $\text{HighBits}(r+a, 2\gamma_2)$.

Voltemos então para o caso do Dilithium. Vimos que o verificador conhece $\hat{\mathbf{w}}$. Então, se valesse que $\|c\mathbf{t}_0\|_\infty < \gamma_2$, então poderíamos enviar um vetor $\mathbf{h}$ de bits, onde cada bit está associado a um coeficiente de cada polinômio, que permitiria

$$\text{HighBits}(\hat{\mathbf{w}} - c\mathbf{t}_0, 2\gamma_2) = \mathbf{w}_1.$$

Porém, caso $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$, não seria possível garantir a validade de um tal vetor de dicas. Então, neste caso, recomeçamos o algoritmo de assinatura com um novo vetor $\mathbf{y}$.

O calculo do vetor binário $\mathbf{h}$ de dicas pode ser calculado usando a seguinte função. Quando $\|c\mathbf{t}_0\|_\infty \geq \gamma_2$, o valor `None` é devolvido para indicar que o algoritmo de assinatura deve ser reiniciado.

```
1   def get_hints_for_w(self, w_and_w1, t0_ntt, s2_ntt, c_ntt):
2
3       w, w1 = w_and_w1
4       ct0 = (t0_ntt * c_ntt).inv_ntt()
5       cs2 = (s2_ntt * c_ntt).inv_ntt()
6       h = self.make_hint(-ct0, w - cs2 + ct0, 2 * self.params.gamma2)
7
8       n_ones_in_h = sum(p.hamming_weight() for p in h)
9       if ct0.norm_infinity() >= self.params.gamma2 or n_ones_in_h > self.params.omega:
10          return None
11
12      assert(self.use_hint(h, w - cs2 + ct0, 2 * self.params.gamma2) == w1)
13
14      return h
```

Note que há também uma outra condição de rejeição, que é se o número de entradas não-nulas em $\mathbf{h}$ for maior do que ω. O valor ω é um parâmetro do esquema calculado de forma que `n_ones_in_h` $\leq \omega$ em 99% dos casos. A vantagem de ter um valor ω fixo é que $\mathbf{h}$ pode ser comprimido se forem enviadas apenas as posições das entradas não-nulas.

As funções auxiliares responsáveis por calcular cada um dos bits do vetor de dicas, dados os coeficientes, têm a seguinte implementação.

```
1   def make_hint_coefficient(self, z, r, alpha):
2       r1 = self.high_bits_coefficient(r, alpha)
3       v1 = self.high_bits_coefficient(r + z, alpha)
4       return int(r1 != v1)
5
6   def make_hint(self, z, r, alpha):
7       v = []
8       for poly_z, poly_r in zip(z, r):
9           v.append([self.make_hint_coefficient(c_z, c_r, alpha)
10                    for c_z, c_r in zip(poly_z, poly_r)])
11      return v
```

Verificação de assinatura: A verificação de assinaturas é análoga à verificação do esquema ilustrado na Figura 1.5. Sua implementação é dada a seguir.

```
def verify(self, pk, message_bytes, signature):
    (rho, t1) = pk
    (c_tilde, z, h) = signature

    ntt_A = self.get_ntt_A_from_seed(rho)

    pk_hash = self.hash_H(rho + t1.as_bytes(), 32)
    mu = self.hash_H(pk_hash + message_bytes, 64)
    c = self.sample_in_ball(c_tilde)

    z_ntt = z.ntt()
    c_ntt = self.ntt_ring.ntt(c)
    t1_ntt = t1.ntt()

    r = self.matrix_poly_vec_product(ntt_A, z_ntt) - ((t1_ntt * c_ntt) * (2**self.
params.d))
    w1 = self.use_hint(h, r.inv_ntt(), 2 * self.params.gamma2)

    if (z.norm_infinity() >= self.params.gamma1 - self.params.beta):
        return False

    if (c_tilde != self.hash_H(mu + w1.as_bytes(), 32)):
        return False

    if sum(p.hamming_weight() for p in h) > self.params.omega:
        return False

    return True
```

Note que, para recalcular $\mathbf{w}_1$, é chamada a função `use_hint` que usa as dicas da forma como mostramos na seção passada. Esta função é definida abaixo.

```
def use_hint(self, h, r, alpha):
    v = []
    for poly_h, poly_r in zip(h, r):
        v.append([self.use_hint_coefficient(c_h, c_r, alpha)
                  for c_h, c_r in zip(poly_h, poly_r)])
    return self.to_poly_vec(v)

def use_hint_coefficient(self, h, r, alpha):
    m = (self.params.q - 1) // alpha
    (r1, r0) = self.decompose(r, alpha)
    if h and r0 > 0:
        return (r1 + 1) % m
    if h and r0 <= 0:
        return (r1 - 1) % m
    return r1
```

1.7. Fundamentos do HQC: códigos lineares

Códigos corretores de erros são muito usados em sistemas de comunicação pois permitirem a recuperação de uma mensagem transmitida mesmo na presença de ruídos causados pelo canal de transmissão. Fixado um alfabeto de comunicação $\mathcal{A}$, que tipicamente será o corpo binário $\mathbb{F}_2$ ou um outro corpo finito $\mathbb{F}_q$. Suponha que queremos transmitir uma mensagem $\mathbf{m} \in \mathcal{A}^k$ de k elementos do nosso alfabeto $\mathcal{A}$. Um bom código corretor de erro deve explicitar dois procedimentos. Primeiro, um algoritmo de codificação, responsável por adicionar redundância a $\mathbf{m}$, obtendo uma palavra de código $\mathbf{c} \in \mathcal{A}^n$. Segundo, um procedimento de decodificação, responsável por reconstruir $\mathbf{m}$ a partir do vetor codificado mesmo que algumas entradas de $\mathbf{c}$ tenham sido corrompidas durante a transmissão.

A Figura 1.6 ilustra o uso de um código de repetição sobre o alfabeto $\mathcal{A} = \mathbb{F}_2$, com

mensagens de tamanho $k = 3$. Podemos ver que o processo de codificação consiste em repetir 3 vezes cada um dos 3 bits de $\mathbf{m}$. Para decodificar, basta verificar, em cada bloco, qual bit aparece mais vezes. É importante notar que, neste código, caso ocorressem dois erros num mesmo bloco, a mensagem seria incorretamente recuperada. Portanto, no pior caso, tal código só é capaz de corrigir um único bit errado.

Figura 1.6. Transmissão de uma palavra através de um canal ruidoso.

Em geral, o processo de codificação pode ser qualquer mapeamento que leva as possíveis mensagens $\mathbf{m} \in \mathcal{A}^k$ em palavras de $\mathcal{A}^n$. Porém, mesmo para valores relativamente pequenos de k o tamanho $\left|\mathcal{A}^k\right|$ é muito grande, e fica inviável guardar uma tal função na memória eficientemente. Isso motiva a definição de códigos lineares, cujo alfabeto é um corpo finito e o procedimento de codificação é eficientemente descrito pela da multiplicação de uma matriz.

Seja p um primo e $q = p^m$, para algum inteiro m. Um código $[n,k]$-linear $\mathcal{C}$ é um subespaço vetorial de dimensão k do espaço $\mathbb{F}_q^n$. Note, portanto, o código $\mathcal{C}$ pode ser descrito tanto como a imagem de uma matriz $\mathbf{G} \in \mathbb{F}_q^{k\times n}$ quanto pelo núcleo de uma matriz $\mathbf{H} \in \mathbb{F}_q^{(n-k)\times n}$. Quaisquer matrizes $\mathbf{G}$ e $\mathbf{H}$ que definem um código $\mathcal{C}$ são ditas matrizes geradoras e de verificação de paridade, respectivamente. Formalmente, verifica-se para tais matrizes que

$$\mathcal{C} = \left\{\mathbf{mG} : \mathbf{m} \in \mathbb{F}_q^k\right\} = \left\{\mathbf{c} \in \mathbb{F}_q^n : \mathbf{cH}^\top = \mathbf{0}\right\}.$$

Dessa forma uma matriz de paridade $\mathbf{H}$ é capaz de identificar se uma palavra faz ou não parte do código $\mathcal{C}$. Isso motiva a definição da síndrome de um vetor $\hat{\mathbf{c}} \in \mathbb{F}_q^n$ como o produto $\mathbf{s} = \hat{\mathbf{c}}\mathbf{H}^\top$. Assim, se a síndrome $\mathbf{s}$ for $\mathbf{0}$, quer dizer que todas as equações lineares induzidas por $\mathbf{H}$ foram satisfeitas e $\hat{\mathbf{c}}$ faz parte do código. Caso contrário, então podemos ver $\hat{\mathbf{c}}$ como a soma de um vetor $\mathbf{c} \in \mathcal{C}$ e um vetor de erro $\mathbf{e} \in \mathbb{F}_q^n$, isto é, $\hat{\mathbf{c}} = \mathbf{c} + \mathbf{e}$. Portanto a síndrome

$$\mathbf{s} = \hat{\mathbf{c}}\mathbf{H}^\top = (\mathbf{c}+\mathbf{e})\,\mathbf{H}^\top = \mathbf{cH}^\top + \mathbf{eH}^\top = \mathbf{eH}^\top$$

pode ajudar na decodificação ao identificar as equações insatisfeitas por conta do erro $\mathbf{e}$.

Tome um vetor $\mathbf{c} \in \mathbb{F}_q^n$, e considere as seguintes definições. Definimos o suporte de $\mathbf{c}$, denotado por $\operatorname{supp}(\mathbf{c})$, como o conjunto de índices de suas entradas não nulas, isto é, $\operatorname{supp}(\mathbf{c}) = \{i : \mathbf{c}[i] \neq 0\}$. O peso de Hamming de $\mathbf{c}$ é definido como o número de elementos de seu suporte, e denotado por $\mathrm{w}(\mathbf{c}) = |\operatorname{supp}(\mathbf{c})|$. A distância de Hamming entre dois vetores $\mathbf{u}$ e $\mathbf{v}$ de $\mathbb{F}_q^n$, denotada por $\operatorname{dist}(\mathbf{u},\mathbf{v})$, é definida como o número de coordenadas em que $\mathbf{u}$ e $\mathbf{v}$ diferem, ou, alternativamente $\operatorname{dist}(\mathbf{u},\mathbf{v}) = \mathrm{w}(\mathbf{u}-\mathbf{v})$.

©2023 Sociedade Brasileira de Computação

1.7.1. Códigos cíclicos e polinômios geradores

Há casos em que códigos é útil considerar códigos em que a dimensão k e o comprimento n são grandes. Nestes casos a representação por matrizes pode ser ineficiente, e, ainda pior, a multiplicação pela matriz geradora necessária para a codificação pode ser muito custosa. Uma subclasse de códigos lineares que cuja representação e codificação são mais eficientes é a de códigos cíclicos, que admitem uma matriz geradora $k \times n$ da forma

$$\mathbf{G} = \begin{bmatrix} 1 & g_1 & g_2 & \cdots & g_{n-k-1} & 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & g_1 & \cdots & g_{n-k-2} & g_{n-k-1} & 1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & g_1 & g_2 & g_3 & \cdots & g_{n-k-1} & 1 \end{bmatrix}.$$

Nota-se que uma tal matriz pode ser representada compactamente somente pelo vetor $\mathbf{g} = [1, g_1, \ldots, g_{n-k-1}, 1]$. Porém, mais importante ainda é o fato de que a codificação $\mathbf{c} = \mathbf{mG}$ de uma mensagem $\mathbf{m} \in \mathbb{F}_q^k$ pode ser computada da seguinte forma. Primeiro, associamos todo vetor $\mathbf{v} = [v_0, v_1, \ldots, v_{m-1}] \in \mathbb{F}_q^m$, ao polinômio $\mathbf{v}(x)$ de forma que

$$\mathbf{v}(x) = v_0 + v_1 x + \ldots + v_{m-1} x^{m-1} = \sum_{i=0}^{m-1} v_i x^i.$$

Seja $\mathbf{g} = [1, g_1, \ldots, g_{n-k-1}, 1] \in \mathbb{F}_q^{n-k+1}$ e $\mathbf{m} \in \mathbb{F}_q^k$ a mensagem que queremos codificar. Então, para o código cíclico gerado por $\mathbf{g}$, vale que

$$\mathbf{c} = \mathbf{mG} \text{ se, e somente se } \mathbf{c}(x) = \mathbf{m}(x)\mathbf{g}(x).$$

Como a multiplicação de polinômios pode ser computada eficientemente através de algoritmos como o de Karatsuba ou da trasformada rápida de Fourier, a codificação de códigos cíclicos é muito eficiente. Exemplos de códigos cíclicos muitos usados são as importantes famílias de códigos BCH e Reed-Solomon.

1.7.2. Decodificação

Há vários modelos de ruídos que podem ocorrer durante a transmissão. Podem ser erros uniformemente aleatórios, ou seguir uma gaussiana, por exemplo. Também há modelos em que coordenadas são apagadas, ou em que erros vêm em rajadas sobre posições contíguas do vetor codificado, como quando um CD é arranhado. Para cada tipo de erro, um diferente código corretor pode ser a melhor opção.

Seja q a potência de um primo e considere o corpo finito $\mathbb{F}_q$. Um modelo genérico e importante de ruído sobre códigos $[n,k]$-lineares sobre $\mathbb{F}_q$ é o canal simétrico q-ário. Este modelo usa um parâmetro $p \in [0,1]$ chamado probabilidade de transição, e gera erros $\mathbf{e} = [e_0, \ldots, e_{n-1}]$ de forma que cada e_i é gerado de forma independente como

$$e_i = \begin{cases} 0 \text{ com probabilidade } (1-p), \\ \text{elemento aleatório de } (\mathbb{F}_q - \{0\}) \text{ com probabilidade } p. \end{cases}$$

Dada uma palavra $\hat{\mathbf{c}} \in \mathbb{F}_q^n$, e um código $[n,k]$-linear $\mathcal{C} \in \mathbb{F}_q$, o problema da decodificação consiste em encontrar a palavra $\mathbf{c} \in \mathcal{C}$ mais próxima de $\hat{\mathbf{c}}$. Embora seja possível construir bons códigos cuja decodificação é eficiente, este problema é NP-difícil [Berlekamp et al. 1978].

O problema da decodificação pode ser formulado da seguinte maneira alternativa, resultando no chamado problema da decodificação por síndrome. Nesta formulação, é dada uma matriz de paridade $\mathbf{H} \in \mathbb{F}_q^{r \times n}$ de um código $\mathcal{C}$ junto com uma síndrome $\mathbf{s} \in \mathbb{F}_q^r$. O problema é encontrar o vetor $\mathbf{e} \in \mathbb{F}_q^n$ de menor peso de Hamming que satisfaz $\mathbf{e}\mathbf{H}^\top = \mathbf{s}$.

1.7.3. Distância mínima de um código

Bons códigos possuem seus elementos a uma boa distância de Hamming uns dos outros, o que garante uma boa capacidade de correção de erros. Isso motiva a definição de distância mínima de um código, que corresponde à menor distância entre duas de suas palavras. Formalmente, a distância mínima de um código linear $\mathcal{C}$, denotada como $d(\mathcal{C})$, pode ser computada como

$$d(\mathcal{C}) = \min_{\mathbf{u},\mathbf{v}\in\mathcal{C}} \operatorname{dist}(\mathbf{u},\mathbf{v}) = \min_{\mathbf{u},\mathbf{v}\in\mathcal{C}} \operatorname{w}(\mathbf{u}-\mathbf{v}).$$

Porém, pela linearidade do código, a diferença dos vetores $\mathbf{a} = (\mathbf{u}-\mathbf{v})$ varre todos os elementos de $\mathcal{C}$. Portanto

$$d(\mathcal{C}) = \min_{\mathbf{u},\mathbf{v}\in\mathcal{C}} \operatorname{w}(\mathbf{u}-\mathbf{v}) = \min_{\mathbf{a}\in\mathcal{C}} \operatorname{w}(\mathbf{a}).$$

Dizemos que um código $\mathcal{C}$ é $[n,k,d]$-linear quando $\mathcal{C}$ é $[n,k]$-linear e $d(\mathcal{C}) = d$. Um tal código poderá decodificar erros de peso até $\lfloor (d-1)/2 \rfloor$.

1.8. HQC

O HQC [Melchor et al. 2021] é um KEM baseado no problema da decodificação de códigos quase cíclicos. Sua construção é muito similar àquela dos esquemas baseados no problema LWE vistos na Seção 1.6, porém com a significativa diferença de trabalhar no corpo binário. Isso faz com que a fase de reconciliação seja mais complicada pois é necessário o uso de códigos corretores de erro poderosos.

Começamos esta seção apresentando os códigos Reed-Muller e Reed-Solomon que, juntos, compõem o código corretor de erros usado para a reconciliação. Em seguida, definimos o problema difícil em que o HQC se baseia. Ao final, o HQC é apresentado em sua versão IND-CPA.

1.8.1. Códigos Reed-Muller

Códigos Reed-Muller são uma família de códigos binários lineares. Para quaisquer inteiros positivos r e m, tais que $0 \le r \le m$, é possível construir um código $[n,k,d]$-linear, denotado $\mathrm{RM}(r,m)$ com as seguintes propriedades. O comprimento do código é $n = 2^m$, sua dimensão é $k = 1 + \binom{m}{1} + \binom{m}{2} + \ldots + \binom{m}{r}$, e sua distância mínima é $d = 2^{m-r}$.

Há várias formas de definir tais códigos, porém, neste trabalho, vamos apresentar diretamente o código Reed-Muller usado no HQC, que usa como parâmetros os valores $(r,m) = (1,7)$. O leitor poderá conferir que tais parâmetros resultam num código

$[128, 8, 64]$-linear. Este código, denotado por $\mathrm{RM}(1,7)$, é gerado pela matriz representada na Figura 1.7.

Figura 1.7. A matriz binária geradora do código RM$(1,7)$.

Para atingir a alta capacidade correção necessária pelo HQC, o código usado usa o código $\mathrm{RM}(1,7)$ repetido M vezes. Pode-se mostrar que, repetindo o código $[128, 8, 64]$-linear M vezes, obtém-se um novo código $[128M, 8, 64M]$-linear. Isto é, há uma expansão linear tanto no tamanho do código e em sua distância mínima, enquanto a dimensão não é alterada.

Como a matriz é do código é relativamente grande, podemos defini-la mais sucintamente em Sage usando uma função auxiliar que converte números inteiros em vetores binários de tamanho fixo.

```
1 def binary_vector_from_int(a, length):
2     bits = bin(a)[2:]
3     v = [0] * length
4     for i, b in enumerate(reversed(bits)):
5         v[i] = int(b)
6     return v
```

Assim, podemos definir os códigos Reed-Muller repetidos usados pelo HQC da seguinte forma.

```
1 class RepeatedReedMullerForHQC():
2 
3     RMCodeLength = 128
4 
5     GeneratorMatrix = matrix([
6         binary_vector_from_int(0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa, RMCodeLength),
7         binary_vector_from_int(0xcccccccccccccccccccccccccccccccc, RMCodeLength),
8         binary_vector_from_int(0xf0f0f0f0f0f0f0f0f0f0f0f0f0f0f0f0, RMCodeLength),
9         binary_vector_from_int(0xff00ff00ff00ff00ff00ff00ff00ff00, RMCodeLength),
10        binary_vector_from_int(0xffff0000ffff0000ffff0000ffff0000, RMCodeLength),
11        binary_vector_from_int(0x00000000ffffffff00000000ffffffff, RMCodeLength),
12        binary_vector_from_int(0x0000000000000000ffffffffffffffff, RMCodeLength),
13        binary_vector_from_int(0xffffffffffffffffffffffffffffffff, RMCodeLength),
14    ])
15 
16    def __init__(self, multiplicity):
17        self.q = 2
18        self.k = 8
19        self.n = 128 * multiplicity
20        self.multiplicity = multiplicity
21        self.encoded_table = [self.encode(binary_vector_from_int(m, self.k))
22                              for m in range(256)]
```

Codificação: A codificação de uma mensagem $\mathbf{m} \in \mathbb{F}_2^8$ pode ser feita simplesmente multiplicando-a pela matriz geradora do código $\mathrm{RM}(1,7)$ e devolvendo o vetor resultante repetido M vezes.

```
1     def encode(self, message):  # Class RepeatedReedMullerForHQC
2         repeated_encoded = (
3             [vector(GF(2), message) * self.GeneratorMatrix] * self.multiplicity)
4         return vector(chain(*repeated_encoded))
```

©2023 Sociedade Brasileira de Computação

Decodificação: Em geral, a decodificação de códigos Reed-Muller pode ser feita eficientemente usando a transformada rápida de Hadamard [Lin and Costello 2004]. Porém, como o código $\mathrm{RM}(1,7)$ tem somente $2^k = 256$ elementos, podemos fazer a decodificação pela distância mínima usando um algoritmo exaustivo.

Para cada possível vetor de $\mathbb{F}_2^{256}$, codifique-o e calcule a distância entre o resultado e palavra ruidosa recebida. Devolva aquele vetor cuja codificação encontra-se à menor distância da palavra recebida.

```
1      def decode(self, noisy_codeword):  # Class RepeatedReedMullerForHQC
2          min_distance = self.n
3          decoded_m = None
4          for m in range(2**8):
5              encoded_m = self.encoded_table[m]
6              distance = (encoded_m - noisy_codeword).hamming_weight()
7              if distance < min_distance:
8                  min_distance = distance
9                  decoded_m = m
10         return binary_vector_from_int(decoded_m, self.k)
```

1.8.2. Códigos Reed-Solomon

Códigos Reed-Solomon são uma importante família de códigos cíclicos corretores de erros. Diferente dos códigos Reed-Muller, códigos Reed-Solomon trabalham com alfabetos q-ários, onde $q = p^m$ é a m-ésima potência de um primo p. Novamente, para deixar a nossa explicação o mais simples e concreta possível, vamos fixar alguns parâmetros para o código usado no HQC, onde $p = 2$, $m = 8$, portanto $q = 256$.

Lembre que, por serem cíclicos, trabalharemos com polinômios para a codificação eficiente. Uma propriedade importante desses códigos é poder garantir a sua capacidade de correção de erros durante sua construção. Seja δ o número de erros que o código deverá ser capaz de corrigir, e seja α um elemento primitivo de $\mathbb{F}_q$. Então o polinômio

$$\mathbf{g}(x) = \prod_{i=1}^{2\delta} (x - \alpha^i) = (x-\alpha)(x-\alpha^2)\dots(x-\alpha^{2\delta}).$$

gera um código $[n,k,d]$-linear de distância mínima $d = 2\delta + 1$, comprimento $n \leq q-1$ e dimensão $k \leq n-d$, chamado Reed-Solomon.[4]

```
1  class ReedSolomonForHQC():
2
3      def __init__(self, n, k):
4          self.n = n  # Block length
5          self.k = k  # Dimension
6          self.q = 256
7          self.delta = (n - k) // 2
8          self.base_field = GF(self.q)
9          self.polynomial_ring = PolynomialRing(self.base_field, 'x')
10         self.x = self.polynomial_ring.gen()
11         self.generator_polynomial = self._compute_generator_polynomial()
12
13     def _compute_generator_polynomial(self):
14         alpha = self.base_field.primitive_element()
15         return prod(self.x - alpha**i for i in range(1, 2*self.delta + 1))
```

[4] Em geral, adota-se $n = q-1$ na definição de códigos Reed-Solomon. Formalmente, os códigos usados pelo HQC são chamados Reed-Solomon encurtados pois $n < q-1$. Como a codificação e decodificação funcionam exatamente da mesma forma, preferimos trabalhar diretamente com os códigos encurtados.

Codificação: Suponha que queremos codificar uma mensagem $\mathbf{m} \in \mathbb{F}_{256}^k$. Lembre que identificamos vetores e polinômios, portanto $\mathbf{m}(x)$ é o polinômio associado à mensagem $\mathbf{m}$. Seja $\mathbf{g}$ o polinômio gerador do código Reed-Solomon $[n,k,d]$-linear. Então a codificação $\mathbf{c} \in \mathbb{F}_{256}^n$ da mensagem $\mathbf{m}$ na forma sistemática é dada por

$$\mathbf{c}(x) = \mathbf{m}(x)x^{n-k} + \left(\mathbf{m}(x)x^{n-k} \bmod \mathbf{g}(x)\right).$$

Considere o vetor codificado $\mathbf{c}$. Note que a mensagem $\mathbf{m}$ pode ser recuperada diretamente dos k coeficientes associados às potências de x^{n-k} até x^{n-1}. A redundância adicionada corresponde aos $n-k$ coeficientes de índice 0 a $(n-k-1)$. Em Sage, podemos implementar a codificação da seguinte forma.

```
1     def encode(self, message):
2         message_polynomial = self.polynomial_ring(message)
3         a = self.x**(self.n - self.k) * message_polynomial
4         b = a % self.generator_polynomial
5         c = b + a
6         return c
```

Note, porém, que os vetores usados no HQC são binários. Então, se quisermos codificar mensagens binárias usando com códigos Reed-Solomon sobre $\mathbb{F}_{256}$, devemos primeiro quebrar a mensagem binária em sequências de 8 bits, e interpretar cada bloco de 8 bits como um elemento de $\mathbb{F}_{256}$, para então codificar. Este procedimento é sintetizado no método abaixo.

```
1     def encode_binary(self, binary_message):
2         assert len(binary_message) == self.k * 8
3         message = [self.base_field(binary_message[i * 8 : (i + 1) * 8])
4                    for i in range(self.k)]
5         encoded = self.encode(message)
6         encoded_binary = chain(*[list(v) for v in encoded])
7         return vector(GF(2), encoded_binary)
```

Decodificação: Quando descrevemos os códigos Reed-Muller na Seção 1.8.1, notamos que o código continha apenas 256 elementos, e, portanto, foi possível usar o algoritmo simples de distância mínima para a decodificação. Infelizmente, os códigos Reed-Solomon usados no HQC contêm $q^k = 256^k$ elementos, que tornam inviável usar a mesma forma geral de decodificação. Nesta seção, vamos descrever um algoritmo eficiente para a decodificação de códigos Reed-Solomon [Lin and Costello 2004].

Suponha que $\mathbf{c}$ é a codificação de uma mensagem $\mathbf{m}$, e seja $\mathbf{r} = \mathbf{c} + \mathbf{e}$ a palavra $\mathbf{c}$ corrompida por um erro $\mathbf{e}$. Considere o polinômio $\mathbf{r}(x) = r_0 + r_1 x + \ldots r_{n-1}x^{n-1}$ associado ao vetor corrompido $\mathbf{r}$. A síndrome do vetor recebido $\mathbf{r}$ em relação ao código Reed-Solomon é dado por $(s_1, \ldots, s_{2\delta})$ onde cada s_i é dado por

$$s_i = \mathbf{r}(\alpha^i) = \mathbf{c}(\alpha^i) + \mathbf{e}(\alpha^i) = \mathbf{e}(\alpha^i).$$

Suponha que sabemos que o o peso de $\mathbf{e}$ é t, isto é, há t entradas não nulas em $\mathbf{e} = [e_0, \ldots, e_{n-1}]$. Digamos que tais entradas são $\operatorname{supp}(\mathbf{e}) = \{j_1, j_2, \ldots, j_t\}$. Então podemos escrever

$$s_i = \sum_{k=1}^{t} e_{j_k}\left(\alpha^{j_k}\right)^k = e_{j_1}\left(\alpha^{j_1}\right)^i + e_{j_2}\left(\alpha^{j_2}\right)^i + \ldots + e_{j_t}\left(\alpha^{j_t}\right)^i. \tag{1}$$

Na forma matricial, temos então o seguinte sistema

$$\begin{bmatrix} \alpha^{j_1} & \dots & \alpha^{j_t} \\ (\alpha^{j_1})^2 & \dots & (\alpha^{j_t})^2 \\ \vdots & \ddots & \vdots \\ (\alpha^{j_1})^{2\delta} & \dots & (\alpha^{j_t})^{2\delta} \end{bmatrix} \begin{bmatrix} e_{j_1} \\ e_{j_2} \\ \vdots \\ e_{j_t} \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_t \end{bmatrix}. \tag{2}$$

Infelizmente, ainda não é possível resolver a equação pois não sabemos nem o número t de erros nem as suas posições $j_1, \dots, j_t$. Considere então o polinômio $\sigma(x)$ definido como

$$\sigma(x) = \left(1 - \alpha^{j_1} x\right) \left(1 - \alpha^{j_2} x\right) \dots \left(1 - \alpha^{j_t} x\right).$$

O polinômio $\sigma(x)$ é chamado localizador de erros pois, através de suas raízes $\alpha^{-j_1}, \dots, \alpha^{-j_t}$, é possível descobrir qual as posições j_i dos erros, já que α é um elemento primitivo de $\mathbb{F}_q$.

Os passos para resolver o sistema da Equação 2 são os seguintes.

1. Descobrir os coeficientes do polinômio $\sigma(x) = 1 + \sigma_1 x + \sigma_2 x^2 + \dots + \sigma_t x^t$.
2. Encontrar as t raízes α^{j_i} de $\sigma(x)$.
3. Obter as posições $j_1, \dots, j_t$, dos erros.
4. Montar a matriz de potências de α e resolver o sistema da Equação 2.

Pela definição de $\sigma(x)$, vale, para todo $k = 1$ até t, que

$$\sigma\left(\alpha^{-j_k}\right) = 1 + \sigma_1 \alpha^{-j_k} + \sigma_2 \left(\alpha^{-j_k}\right)^2 + \dots + \sigma_t \left(\alpha^{-j_k}\right)^t = 0.$$

A ideia é relacionar a equação acima com as entradas $(s_1, \dots, s_{2\delta})$ da síndrome. Compare a equação acima com a Equação 1 que define os valores s_i. Apesar de haver potências de α na equação, as potências são negativas. Além disso, faltam os fatores e_j.

Fixe então um ℓ qualquer tal que $1 \leq \ell \leq t$. Ao multiplicar a equação por $e_{j_k} \alpha^{j_k(\ell+t)}$, obtemos

$$\begin{aligned} 0 &= e_{j_k} \alpha^{j_k(\ell+t)} \left(1 + \sigma_1 \alpha^{-j_k} + \sigma_2 \left(\alpha^{-j_k}\right)^2 + \dots + \sigma_t \left(\alpha^{-j_k}\right)^t\right) \\ &= e_{j_k} \alpha^{j_k(\ell+t)} + \sigma_1 e_{j_k} \alpha^{j_k(\ell+t-1)} + \sigma_2 e_{j_k} \alpha^{j_k(\ell+t-2)} + \dots + \sigma_t e_{j_k} \alpha^{j_k(\ell)}. \end{aligned}$$

Como a equação acima vale para qualquer k, podemos somar as t equações para $1 \leq k \leq t$ e obtemos

$$\begin{aligned} 0 &= \sum_{k=1}^{t} \left(e_{j_k} \alpha^{j_k(\ell+t)} + \sigma_1 e_{j_k} \alpha^{j_k(\ell+t-1)} + \dots + \sigma_t e_{j_k} \alpha^{j_k(\ell)}\right) \\ &= \sum_{k=1}^{t} \left(e_{j_k} \alpha^{j_k(\ell+t)}\right) + \sum_{k=1}^{t} \left(\sigma_1 e_{j_k} \alpha^{j_k(\ell+t-1)}\right) + \dots + \sum_{k=1}^{t} \left(\sigma_t e_{j_k} \alpha^{j_k(\ell)}\right) \\ &= \sum_{k=1}^{t} \left(e_{j_k} \alpha^{j_k(\ell+t)}\right) + \sigma_1 \sum_{k=1}^{t} \left(e_{j_k} \alpha^{j_k(\ell+t-1)}\right) + \dots + \sigma_t \sum_{k=1}^{t} \left(\sigma_t e_{j_k} \alpha^{j_k(\ell)}\right). \end{aligned}$$

©2023 Sociedade Brasileira de Computação

Lembre que, pela Equação 1, vale que $s_i = \sum_{k=1}^{t} \left(e_{j_k} \alpha^{j_k i}\right)$. Podemos então substituir cada somatória pelo respectivo valor de síndrome, reduzindo a equação acima a

$$s_{\ell+t} + \sigma_1 s_{\ell+t-1} + \ldots + \sigma_t s_\ell = 0.$$

Como tal equação vale para todo $1 \leq \ell \leq t$, obtemos o seguinte sistema

$$\begin{bmatrix} s_t & s_{t-1} & \cdots & s_1 \\ s_{t+1} & s_t & \cdots & s_2 \\ \vdots & \vdots & \ddots & \vdots \\ s_{2t-1} & s_{2t-2} & \cdots & s_t \end{bmatrix} \begin{bmatrix} \sigma_1 \\ \sigma_2 \\ \vdots \\ \sigma_t \end{bmatrix} + \begin{bmatrix} s_{t+1} \\ s_{t+2} \\ \vdots \\ s_{2t} \end{bmatrix} = \mathbf{0}. \quad (3)$$

Note que, apesar de sabermos cada s_i, para $1 \leq i \leq 2\delta$, nós não sabemos o valor de t. Neste momento, isso impede que possamos construir tal sistema e recuperar os coeficientes σ_j do polinômio localizador de erros $\sigma(x)$.

Há duas saídas. A primeira é simples: testar diferentes valores de t, começando em δ. Para cada valor de t, constrói-se o sistema correspondente e tenta-se resolvê-lo. Caso seja possível, o valor foi encontrado. Caso não seja, tentamos o valor $t \leftarrow (t-1)$, até chegar em $t = 0$. Apesar da busca exaustiva por t funcionar bem para casos pequenos, esse algoritmo pode ser muito ineficiente para códigos Reed-Solomon maires, devido ao grande número de sistemas lineares que devem ser resolvidos.

Uma alternativa mais eficiente é notar que este sistema linear tem uma forma especial, que consiste num *linear-feedback shift register*. Pode-se então usar o algoritmo de Berlekamp-Massey para encontrar ao mesmo tempo o menor t que satisfaz a equação e os coeficientes de $\sigma(x)$. Nosso código usa esta opção, porém, devido a limitações de espaço, não é possível explicar aqui o funcionamento desse algoritmo.

Lembre que o código Reed-Solomon usado pelo HQC usa $q = 256$. Dessa forma, podemos calcular as raízes de σ simplesmente testando para quais, dos 256 elementos $\beta \in \mathbb{F}_q$ possíveis, vale que $\sigma(\beta) = 0$. Com isso, conseguimos construir o sistema linear e, se possível, resolvê-lo. O código abaixo é responsável por este procedimento.

```
1   def solve_error_linear_system(self, syndromes, error_location_poly):
2       alpha = self.base_field.primitive_element()
3       alpha_log_table = {alpha**i: i for i in range(1, self.n + 1)}
4       roots = [a for a in self.base_field if error_location_poly(a) == 0]
5       error_betas = [e.inverse() for e in roots]
6       error_positions = [alpha_log_table[b] for b in error_betas]
7
8       beta_matrix = matrix(GF(self.q), [
9           [beta**i for beta in error_betas] for i in range(1, 2*self.delta + 1)])
10      error_values = beta_matrix.solve_right(vector(syndromes))
11      error = self.polynomial_ring(
12                  sum(e * self.x**i for i, e in zip(error_positions, error_values)))
13      return error
```

Podemos então descrever totalmente o algoritmo de decodificação da seguinte forma. Note que, caso haja problemas ao construir ou resolver o sistema de valores de erros, há uma falha de decodificação e o valor None é devolvido.

```
1   def decode(self, noisy_codeword):
```

```
2         alpha = self.base_field.primitive_element()
3         syndromes = [noisy_codeword(alpha**i) for i in range(1, 2*self.delta + 1)]
4         error_location_poly = self.berlekamp_massey(syndromes)
5
6         try:
7             error = self.solve_error_linear_system(syndromes, error_location_poly)
8             codeword_polynomial = noisy_codeword - error
9             return self.polynomial_ring(list(codeword_polynomial)[-self.k:])
10
11        except (KeyError, ValueError):
12            return None
```

Implementamos também, de forma análoga à codificação, a interface em binário para a decodificação.

```
1     def decode_binary(self, noisy_binary_codeword):
2         assert len(noisy_binary_codeword) == self.n * 8
3
4         noisy_codeword = [self.base_field(noisy_binary_codeword[i * 8 : (i + 1) * 8])
5                           for i in range(self.n)]
6
7         decoded = self.decode(self.polynomial_ring(noisy_codeword))
8         decoded_binary = chain(*[list(v) for v in decoded])
9         return vector(GF(2), decoded_binary)
```

1.8.3. Concatenando os códigos Reed-Muller e Reed-Solomon

A Figura 1.8 ilustra como é feita a concatenação de códigos Reed-Muller e Reed-Solomon. Estes códigos concatenados podem ser implementados da seguinte forma. Note como a classe abaixo serve apenas como um orquestrador de chamadas aos códitos Reed-Muller e Reed-Solomon.

```
1 class RMRSCodeForHQC():
2
3     def __init__(self, rs_n, rs_k, rm_multiplicity, n=None):
4         self.rs_code = ReedSolomonForHQC(rs_n, rs_k)
5         self.rm_code = RepeatedReedMullerForHQC(rm_multiplicity)
6
7         if n == None:
8             self.padding_length = 0
9         else:
10            self.padding_length = n - self.rm_code.n * self.rs_code.n
11
12    def encode(self, binary_message):
13        rs_codeword = self.rs_code.encode_binary(binary_message)
14        rm_messages = [rs_codeword[i * 8 : (i + 1) * 8] for i in range(self.rs_code.n)]
15        rm_codewords = [self.rm_code.encode(m) for m in rm_messages]
16        return vector(GF(2), chain(*rm_codewords, [0] * self.padding_length))
17
18    def decode(self, noisy_codeword):
19        # By the way rm_noisy_codewords is defined, the padding is naturally removed
20        rm_noisy_codewords = [noisy_codeword[i * self.rm_code.n : (i + 1) * self.rm_code
      .n]
21                              for i in range(self.rs_code.n)]
22        rm_codewords = [self.rm_code.decode(c) for c in rm_noisy_codewords]
23        rs_noisy_codeword = vector(chain(*rm_codewords))
24        return self.rs_code.decode_binary(rs_noisy_codeword)
```

1.8.4. O problema da decodificação por síndrome para códigos quase cíclicos

O principal desafio em usar problemas de Teoria de Códigos para construir esquemas criptográficos é o baixo desempenho inerente aos códigos necessários. Tome o problema da decodificação por síndrome, por exemplo. Como a decodificação deve ser

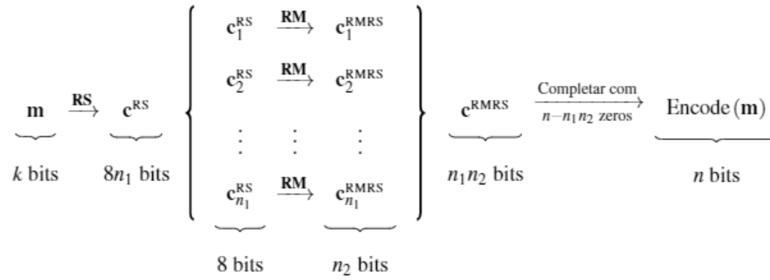

Figura 1.8. O processo de codificação usando códigos Reed-Muller e Reed-Solomon concatenados.

difícil para quem não tem a chave secreta, os códigos em questão devem ser aleatórios ou sua estrutura secreta deve estar muito bem escondida.

Dessa forma, para atingir níveis altos de segurança, os parâmetros $[n,k]$ devem ser relativamente grandes, fazendo com que as matrizes, tanto a geradora quanto a de paridade, ocupem muito espaço na memória e operações típicas como a multiplicação de matriz por vetor seja muito ineficiente. Esquemas modernos baseados em Teoria de Códigos, como o BIKE [Aragon et al. 2022] e o próprio HQC [Melchor et al. 2021], são muito eficientes por usarem códigos chamados quase-cíclicos, que permitem representação e operações mais eficientes.

Antes de vermos a definição de códigos quase cíclicos, primeiro lembre que uma matriz circulante definida por um vetor $\mathbf{v} = [v_0, \ldots, v_{n-1}]$ é a matriz

$$\operatorname{rot}(\mathbf{v}) = \begin{bmatrix} v_0 & v_{n-1} & \cdots & v_1 \\ v_1 & v_0 & \cdots & v_2 \\ \vdots & \vdots & \ddots & \vdots \\ v_{n-1} & v_{n-2} & \cdots & v_0 \end{bmatrix}.$$

O conjunto de matrizes $n \times n$ circulantes com entradas em $\mathbb{F}_q$ forma um anel. Em particular, soma e multiplicação de matrizes circulantes resulta em matrizes também circulantes.

Códigos quase-cíclicos são códigos que admitem matrizes geradoras formadas por blocos de matrizes circulantes. Assim, um código quase-cíclico de índice c admite uma matriz geradora sistemática da forma

$$\mathbf{G} = \begin{bmatrix} \mathbf{I} & \operatorname{rot}(\mathbf{g}_1) & \cdots & \operatorname{rot}(\mathbf{g}_{c-1}) \end{bmatrix} \in \mathbb{F}_2^{n \times cn}.$$

De forma equivalente, um código quase-cíclico admite uma matriz de paridade da forma

$$\mathbf{H} = \begin{bmatrix} \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} & \operatorname{rot}(\mathbf{h}_1) \\ \mathbf{0} & \mathbf{I} & \cdots & \mathbf{0} & \operatorname{rot}(\mathbf{h}_2) \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{I} & \operatorname{rot}(\mathbf{h}_{c-1}) \end{bmatrix} \in \mathbb{F}_2^{(cn-n) \times cn},$$

para vetores $\mathbf{h}_i \in \mathbb{F}_2^n$.

Definição 6 (Problema da decodificação por síndrome para códigos quase-cíclicos)**.** São dados inteiros n, w e c. Escolha aleatoriamente, de maneira uniforme, uma matriz de paridade $\mathbf{H} \in \mathbb{F}_2^{(cn-n)\times cn}$ de um código $[cn,n]$-linear quase-cíclico de índice c. Escolha um vetor esparso $\mathbf{x} = (\mathbf{x}_0,\ldots,\mathbf{x}_{c-1})$ aleatoriamente do conjunto $\mathbb{F}_2^{cn}$ de forma que o peso de cada $\mathbf{x}_i \in \mathbb{F}_2^n$ seja igual a w. Calcule a síndrome $\mathbf{s} = \mathbf{x}\mathbf{H}^\top$. Então, dados $(\mathbf{H},\mathbf{s})$, o problema da decodificação por síndrome pede para encontrar um vetor $\mathbf{y} = (\mathbf{y}_0,\ldots,\mathbf{y}_{c-1}) \in \mathbb{F}_2^{cn}$, tal que $\mathbf{y}\mathbf{H}^\top = \mathbf{s}$ e o peso de cada $\mathbf{y}_i$ seja $\mathrm{w}(\mathbf{y}_i) = w$.

□

1.8.5. Implementação do HQC

Nesta seção, apresentamos o HQC e sua implementação. Veremos que, em alto nível, sua construção é similar à de esquemas baseados no LWE, como o Kyber. Porém, por trabalhar com vetores binários, a codificação e decoficação são significativamente mais complicadas, exigindo códigos corretores de erros poderosos.

Seleção de parâmetros e inicialização: Dado o nível de segurança λ desejado, são selecionados os parâmetros $n_1,n_2,M,n,k,w,w_{\mathbf{r}},w_{\mathbf{e}}$. Estes parâmetros são responsáveis por definir o código concatenado formado pelos códigos Reed-Solomon e Reed-Muller repetido. O código Reed-Solomon usado será $[n_1,k/8]$-linear sobre $\mathbb{F}_{256}$, enquanto o código Reed-Muller repetido será $[n_2,8]$-linear. Os parâmetros $w,w_{\mathbf{r}}$ e $w_{\mathbf{e}}$ correspondem ao peso dos vetores esparsos usados durante a geração de chaves e encriptação.

O código a seguir é responsável pela inicialização dos parâmetros usados no HQC.

```
1  class HQC_PKE_CPA():
2      SecurityParameters = {
3          128: HQCParameters(n1=46, n2=384, multiplicity=3, n=17669,
4                             k=128, w=66, w_r=77, w_e=77),
5          192: HQCParameters(n1=56, n2=640, multiplicity=5, n=35851,
6                             k=192, w=100, w_r=114, w_e=114),
7          256: HQCParameters(n1=90, n2=640, multiplicity=5, n=57637,
8                             k=256, w=133, w_r=149, w_e=149),
9      }
10     def __init__(self, security_level):
11         self.params = self.SecurityParameters[security_level]
12         self.rmrs = RMRSCodeForHQC(rs_n=self.params.n1, rs_k=self.params.k // 8,
13                                    rm_multiplicity=self.params.multiplicity,
14                                    n=self.params.n)
```

Geração de chaves: Escolha um vetor $\mathbf{h}$ de n bits aleatoriamente de $\mathbb{F}_2^n$. Escolha vetores esparsos $\mathbf{x}$ e $\mathbf{y}$ aleatoriamente do conjunto $\left\{\mathbf{v} \in \mathbb{F}_2^n : \mathrm{w}(\mathbf{z}) = w\right\}$. Calcule o vetor $\mathbf{s} = \mathbf{x} + \mathbf{y} \cdot \mathbf{h}$. As chaves pública e secreta serão $\mathbf{pk} = (\mathbf{s},\mathbf{h})$ e $\mathbf{sk} = (\mathbf{x},\mathbf{y})$, respectivamente. Note que a chave secreta está protegida pelo problema da decodificação por síndrome.

```
1      def generate_binary_vector_of_fixed_weight(self, xof, weight):
2          support = xof_sample_k_indexes(xof, self.params.n, weight)
3          v = zero_vector(GF(2), self.params.n)
4          for s in support:
5              v[s] = 1
6          return v
7
8      def vector_product(self, a, b):
9          R = PolynomialRing(GF(2), 'x')
```

```
10          x = R.gen()
11          Q = R.quotient(x**self.params.n - 1)
12          return vector(GF(2), Q(list(a)) * Q(list(b)))
13
14      def keygen(self, randomness):
15          xof = SHAKE256.new(randomness)
16          h = random_vector(GF(2), self.params.n)
17          x = self.generate_binary_vector_of_fixed_weight(xof, self.params.w)
18          y = self.generate_binary_vector_of_fixed_weight(xof, self.params.w)
19          s = x + self.vector_product(h, y)
20
21          return (h, s), (x, y)
```

Encriptação: Seja $\mathbf{m} \in \mathbb{F}_2^k$ a mensagem a ser encriptada e $(\mathbf{s}, \mathbf{h})$ a chave pública do destinatário. Primeiro, escolha dois vetores esparsos $\mathbf{r}_1$ e $\mathbf{r}_2$ aleatoriamente do conjunto $\left\{\mathbf{z} \in \mathbb{F}_2^n : \mathrm{w}(\mathbf{z}) = w_\mathbf{r}\right\}$. Similarmente, escolha um outro vetor esparso $\mathbf{e}$ aleatoriamente de $\left\{\mathbf{z} \in \mathbb{F}_2^n : \mathrm{w}(\mathbf{z}) = w_\mathbf{e}\right\}$. Calcule os vetores $\mathbf{u} = \mathbf{r}_1 + \mathbf{r}_2 \cdot \mathbf{h}$ e $\mathbf{v} = \mathrm{Encode}(\mathbf{m}) + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$. O texto encriptado é dado por $\mathbf{c} = (\mathbf{u}, \mathbf{v})$.

```
1     def encrypt(self, pk, message, randomness):
2         h, s = pk
3         xof = SHAKE256.new(randomness)
4         e = self.generate_binary_vector_of_fixed_weight(xof, self.params.w_e)
5         r_1 = self.generate_binary_vector_of_fixed_weight(xof, self.params.w_r)
6         r_2 = self.generate_binary_vector_of_fixed_weight(xof, self.params.w_r)
7         u = r_1 + self.vector_product(h, r_2)
8         v = self.rmrs.encode(message) + self.vector_product(s, r_2) + e
9         return (u, v)
```

Decriptação: Dado um texto cifrado $\mathbf{c} = (\mathbf{u}, \mathbf{v})$ e a chave secreta $\mathbf{x}, \mathbf{y}$, primeiro calcule o vetor $\mathbf{c}' = \mathbf{v} + \mathbf{u} \cdot \mathbf{y}$. Note que este vetor é

$$\begin{aligned}
\mathbf{c}' &= \mathbf{mG} + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e} + (\mathbf{r}_1 + \mathbf{r}_2 \cdot \mathbf{h}) \cdot \mathbf{y} \\
&= \mathbf{mG} + (\mathbf{x} + \mathbf{y} \cdot \mathbf{h}) \cdot \mathbf{r}_2 + \mathbf{e} + (\mathbf{r}_1 + \mathbf{r}_2 \cdot \mathbf{h}) \cdot \mathbf{y} \\
&= \mathbf{mG} + \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{r}_1 \cdot \mathbf{y} + \mathbf{e}.
\end{aligned}$$

Como os vetores $\mathbf{x}, \mathbf{y}, \mathbf{r}_1, \mathbf{r}_2$, e $\mathbf{e}$ são todos esparsos, então é esperado que o vetor de erro $\mathbf{e}' = \mathbf{x} \cdot \mathbf{r}_2 + \mathbf{r}_1 \cdot \mathbf{y} + \mathbf{e}$ também seja relativamente esparso. Dessa forma, o código RMRS pode-se facilmente decodificar o erro $\mathbf{e}'$ e recuperar a mensagem $\mathbf{m}$.

```
1     def decrypt(self, sk, ciphertext):
2         u, v = ciphertext
3         x, y = sk
4         c_prime = v - self.vector_product(u, y)
5
6         return self.rmrs.decode(c_prime)
```

1.9. Conclusão

Neste capítulo, mostramos detalhes da construção de esquemas criptográficos pós-quânticos. Embora os esquemas apresentem certas similaridades entre si, por abordarmos aspectos teóricos e práticos, pudemos tratar de uma coleção abrangente de temas atuais em criptografia. Com o NTRU, que é o esquemas mais antigo entre os apresentados, vimos como o Sage pode facilitar a experimentação com a prototipagem de esquemas

algébricos. Com o Kyber, apresentamos detalhes da Transformada da Teoria dos Números e vimos como ela é usada em esquemas modernos. Ao apresentar o Dilithium, discutimos a transformação de Fiat-Shamir, e a importância da amostragem por rejeição para proteger segredos. Por fim, o HQC nos permitiu discutir teoria de códigos corretores de erros, mostrando como implementar códigos Reed-Muller e Reed-Solomon em Sage e seu uso em criptografia.

É importante notar que este capítulo não substitui as especificações dos esquemas apresentados. Em particular, há questões que não puderam ser tratadas com maior profundidade por falta de espaço. Alguns pontos importantes de que não tratamos são os ataques contra os esquemas e os motivos por trás da escolha de parâmetros seguros. Há ainda questões mais profundas sobre os perigos das falhas de decriptação, e como podemos garantir que a probabilidade de falha seja desprezável. Além disso, este material não discute dificuldades de implementação segura de esquemas criptográficos nem ataques de canal lateral.

No entanto, esperamos que nosso material possa contribuir para que essa área de pesquisa seja mais acessível. Em particular, a implementação que disponibilizamos ao público poderá ser útil a novos pesquisadores interessados tanto em prototipar soluções mais eficientes quando em criptanálise.

Referências

[Alagic et al. 2022] Alagic, G., Apon, D., Cooper, D., Dang, Q., Dang, T., Kelsey, J., Lichtinger, J., Miller, C., Moody, D., Peralta, R., et al. (2022). Status report on the third round of the NIST post-quantum cryptography standardization process. Technical report, National Institute of Standards and Technology.

[Aragon et al. 2022] Aragon, N., Barreto, P. S. L. M., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.-C., Gaborit, P., Ghosh, S., Gueron, S., Güneysu, T., Aguilar-Melchor, C., Misoczki, R., Persichetti, E., Richter-Brockmann, J., Sendrier, N., Tillich, J.-P., Vasseur, V., and Zémor, G. (2022). BIKE: Bit flipping key encapsulation. https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf.

[Avanzi et al. 2019] Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehlé, D. (2019). CRYSTALS-Kyber: Algorithm specifications and supporting documentation (2020). *NIST PQC Round*, 2(4):1–43.

[Bai et al. 2021] Bai, S., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., and Stehlé, D. (2021). CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (version 3.1). *NIST Post-Quantum Cryptography Standardization Round*, 3.

[Berlekamp et al. 1978] Berlekamp, E. R., McEliece, R. J., and Van Tilborg, H. C. (1978). On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386.

[Bernstein et al. 2019a] Bernstein, D. J., Chou, T., Lange, T., Misoczki, R., Niederhagen, R., Persichetti, E., Schwabe, P., Szefer, J., and Wang, W. (2019a). Classic McEliece: conservative code-based cryptography. *NIST submissions*.

[Bernstein et al. 2019b] Bernstein, D. J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., and Schwabe, P. (2019b). The SPHINCS+ signature framework. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 2129–2146.

[Chen et al. 2020] Chen, C., Danba, O., Hoffstein, J., Hülsing, A., Rijneveld, J., Schanck, J. M., Saito, T., Schwabe, P., Whyte, W., Xagawa, K., Yamakawa, T., and Zhang, Z. (2020). NTRU: Algorithm specifications and supporting documentation. *Brown University and Onboard security company, Wilmington USA*.

[Chen et al. 2016] Chen, L., Jordan, S., Liu, Y.-K., Moody, D., Peralta, R., Perlner, R. A., and Smith-Tone, D. (2016). *Report on post-quantum cryptography*, volume 12. US Department of Commerce, National Institute of Standards and Technology.

[D'Anvers et al. 2018] D'Anvers, J.-P., Karmakar, A., Sinha Roy, S., and Vercauteren, F. (2018). Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM. In *Progress in Cryptology–AFRICACRYPT 2018: 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7–9, 2018, Proceedings 10*, pages 282–305. Springer.

[Hoffstein et al. 1998] Hoffstein, J., Pipher, J., and Silverman, J. H. (1998). NTRU: A ring-based public key cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer.

[Hofheinz et al. 2017] Hofheinz, D., Hövelmanns, K., and Kiltz, E. (2017). A modular analysis of the Fujisaki-Okamoto transformation. In *Theory of Cryptography Conference*, pages 341–371. Springer.

[Lin and Costello 2004] Lin, S. and Costello, D. J. (2004). *Error Control Coding*. Pearson Education.

[Melchor et al. 2021] Melchor, C. A., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Bos, J., Deneuville, J.-C., Dion, A., Gaborit, P., Lacan, J., Persichetti, E., Robert, J.-M., Véron, P., and Zémor, G. (2021). Hamming Quasi-Cyclic: HQC. https://pqc-hqc.org/doc/hqc-specification_2021-06-06.pdf.

[Peikert 2015] Peikert, C. (2015). A decade of lattice cryptography. Cryptology ePrint Archive, Paper 2015/939. https://eprint.iacr.org/2015/939.

[Prest et al. 2020] Prest, T., Fouque, P.-A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., and Zhang, Z. (2020). Falcon. *Post-Quantum Cryptography Project of NIST*.

[Shor 1994] Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE.

©2023 Sociedade Brasileira de Computação