Segurança em Computação – Trabalho Prático: Classic McEliece

Gustavo Emanuel Kundlatsch[1]

[1]Departamento de Informática e Estatística –
Universidade Federal de Santa Catarina (UFSC)

Entrega 1

Proposta de Trabalho

O algoritmo que foi sorteado para o trabalho, e que será seguido, é o Classic McEliece. A principal fonte para o desenvolvimento das demais entregas será o *website* do algoritmo [Bernstein], que armazena as três submissões feitas ao concurso do NIST, com o código e os documentos (guias e especificações) atrelados.

Para as demais entregas, as propostas de trabalho são as seguintes:

Doc2 Respostas para as oito questões requisitadas para a entrega, utilizando os documentos fornecidos pelos autores nas submissões ao NIST. Será dado um enfoque nos parâmetros que o algoritmo pode receber, uma vez que essa é a principal maneira de garantir que ele tenha robustes em um cenário de criptografia pós-quântica, e nos ataques clássicos utilizados contra o McEliece, pois é um sistema bastante antigo e diversos métodos de ataque já foram propostos.

Doc3 Texto explicando o funcionamento do algoritmo, utilizando a ferramenta *libmcleece*, que implementa os algoritmos de geração de chave, criptografar e descriptografar do Classic McEliece. Os principais pontos do código serão utilizados para explicar o funcionamento dos algoritmos.

Vídeo Produção de um vídeo expositivo com duração de 10 a 20 minutos seguindo as explicações, exemplos e demonstração do Doc3.

Entrega 2

Em que problema matemático o algoritmo é baseado? Apresente um exemplo

O algoritmo é baseado na dificuldade de resolução de códigos lineares, um problema que é provado ser NP-Hard. Na teoria de códigos, um código linear é um código de correção de erros para o qual qualquer combinação linear de code words também é uma code word. Nesse contexto, uma code word é um elemento de uma codificação ou protocolo padronizado. Cada code word é montada de acordo com as regras específicas de sua codificação e a ela é atribuído um significado único.

A definição formal de código linear é: Um código linear de comprimento n e posto k é um subespaço linear C com dimensão k do espaço vetorial $\mathbb{F}_q^n$ onde $\mathbb{F}_q$ é o corpo finito com q elementos. Esse código é chamado de código q-ário. Se q = 2 ou q = 3, o código é descrito como um código binário ou um código ternário, respectivamente. Os vetores em C são chamados de palavras-código. O tamanho de um código é o número de palavras-código e é igual a q^k. O peso de uma palavra-código é o número de seus elementos que são diferentes de zero e a distância entre duas palavras-código é a distância de Hamming entre elas, ou seja, o número de elementos em que diferem. A distância d do código linear é o peso mínimo de suas palavras-código diferentes de zero ou, de forma equivalente, a distância mínima entre palavras-código distintas. Um código linear de comprimento n, dimensão k e distância d é chamado de código [n, k, d].

Um exemplo de código linear são os códigos de Hamming, amplamente utilizados em sistemas de comunicação digital. Essa classe de códigos de correção de erros foi inventada por Richard Hamming no começo da década de 40, e utiliza o conceito de bit de paridade para repartir um bloco de bits em áreas e fazer verificações para detectar e corrigir possiveis erros [3Blue1Brown]. Alternativemente, uma matriz de Hamming pode ser construida para que o código de correção de erros possa ser computado com técnicas de álgebra linear. Para isso, uma matriz de geração (G) e uma matriz de paridade (H) são construidas com os dados a serem verificados (ambas matrizes binárias, onde todas as entradas são zero ou um). O produto da matriz de paridade pela matriz de geração transposta deve ser igual a uma matriz onde todas as entradas são zero, para que não hajam erros.

Como este problema foi modificado de forma a permitir a criação de um algoritmo de chave-pública pós-quântico?

O trabalho que apresentou originalmente o McEliece é de 1978. Mesmo após 40 anos de desenvolvimento da área de segurança, o algoritmo se mostrou sólido e estável ao longo das décadas. Isso porque o algoritmo original possui parêmetros que foram definidos pensando apenas para a segurança 2^64, mas como esses parâmetros são variáveis, diversas propostas para esses valores foram feitas pelos autores que o submeteram ao concurso do NIST. O problema matemático em que o algoritmo se baseia foi mantido, e as mudanças para o tornar seguro para um cenário pós-quântico focam na alteração dos parâmetros e alguns outros aspectos, como um KEM (Key encapsulation mechanism).

Mais detalhes sobre os parâmetros do algoritmo estão disponíveis no anexo A, ao final do documento.

Quais observações foram feitas nas rodadas anteriores em relação a este algoritmo?

Na segunda rodada, o NIST requisitou mais níveis de segurança, e algumas variantes dos parâmetros apresentados na submissão original foram apresentadas. Essa submissão também incluiu uma alternativa para a geração de chaves mais complicado mas mais rápido, como uma proposta de modificação para ser avaliada.

Na terceira rodada, os parâmetros propostos na segunda foram incorporados ao trabalho, e o método de geração de chaves foi alterado, com o objetivo de modularizar a corretude e confiabilidade do mapa utilizado (da entrada ao texto cifrado) e a segurança da fonte de bytes aleatórios.

Quais ataques ou falhas de segurança foram apresentados (incluindo as rodadas anteriores)?

Existem diversas tentativas de quebrar a criptogravia do McEliece, uma vez que é um algoritmo bastante antigo. Na primeira submissão para o NIST, os autores realizaram um apanhado das principais táticas.

Os ataques mais rápidos conhecidos usam decodificação de conjunto de informações (information-set decoding ou ISD). A forma mais simples de ISD tenta adivinhar um conjunto de informações livre de erros no código. Um conjunto de informações é, por definição, um conjunto de posições que determina uma palavra-código inteira. O conjunto está livre de erros, por definição, se evitar todas as posições de erro na "palavra recebida", ou seja, o texto cifrado; então, o texto cifrado nessas posições é exatamente a palavra-código nessas posições. O invasor determina o resto da palavra-código por álgebra linear e vê se o ataque foi bem-sucedido verificando se o peso do erro é correto. No entanto, a chance de o conjunto estar livre de erros cai rapidamente à medida que o número de erros aumenta, e por conta disso a segurança do McEliece com os parâmetros apresentados nessa submissão é considerada alta. Além disso, na prática, esse tipo de ataque possui um gargalo que é a quantidade de memória RAM que precisa, pois são realizados acessos aleatórios a um array gigantesco, muito maior que a chave publica sendo utilizada. Segundo os autores, a mudança de parâmetros é o suficiente até mesmo para evitar ataques de computadores quânticos.

Uma estratégia de inversão diferente é encontrar a chave privada, não por força bruta, mas sabendo o polinômio $\mathbb{F}_q[x]$ g, descobrir os valores subsequentes da chave ou vice-versa. Porém, o número de escolhas para a combinação mais fraca de parâmetros do algoritmo é 2^{1600}. Em um cenário de ataque com várias mensagens, o custo de encontrar a chave privada é distribuído por várias mensagens. Existem também ataques de várias mensagens mais rápidos que não dependem da localização da chave privada. Em vez de analisar a segurança de várias mensagens em detalhes. Os autores da submissão acreditam que o ataque a n alvos não tem um ganho maior do que um fator n. Segundo eles, o nível de segurança do McEliece alto o suficiente para que isso não seja um problema para qualquer valor rasoável de n.

Por fim, existem os ataques de texto cifrado escolhido (chosen-ciphertext), um modelo de ataque para criptanálise em que o criptanalista pode coletar informações ao obter as descriptografias dos textos cifrados escolhidos. A partir dessas informações, o

adversário pode tentar recuperar a chave secreta oculta usada para descriptografar. Contra o sistema McEliece tradicionalmente é adicionado dois erros a um texto cifrado Gm + e, sendo e uma sequência de bits de comprimento n e peso de Hamming t (onde t é a capacidade garantida de correção de erros, um dos parâmetros do algoritmo). Isso é equivalente a adicionar dois erros a e. A criptografia é bem-sucedida se e somente se o vetor de erro resultante tiver peso t, ou seja, exatamente uma das duas posições de erro já estava em e. Esse tipo de ataque não funciona por conta do KEM modificado que os autores desenvolveram para a submissão, que adiciona um hash em e (que não tem como ser computado pelo atacante).

Outros ataques ou falhas de segurança não foram apontados, mas os autores foram requisitados a aumentar a segurança do modelo, e por isso algumas escolhas de parâmetros foram alteradas como descrito na pergunta sobre modificações do algoritmo.

Quais modificações foram realizadas para resolver os problemas apontados?

Como foi descrito anteriormente, a escolha de parâmetros mais robustos para o algoritmo clássico e a implementação de um KEM mais seguro resolve o problema dos ataques conhecidos.

Quais são as vantagens e desvantagens em relação aos outros algoritmos concorrendo no concurso de padronização do NIST?

O Classic McEliece é o único algoritmo dos finalistas que é baseado em código. Algoritmos de criptografia pós-quântica podem ser geralmente categorizados em quatro tipos [Relyea]:

- **Criptografia baseada em hash:** Nesse tipo de algoritmo, um número aleatório é gerado para cada cada assinatura possível, então esse valor é passado por uma função de hash e publicado. As assinaturas são esses números aleatórios.
- **Criptografia baseada em código:** O McEliece foi o primeiro algoritmo de criptografia desse tipo. Engloba todos os algoritmos que tem uma abordagem de verificação de erros em uma dada matriz.
- **Criptografia baseada em reticulados (lattice):** Reticulados são conjuntos de pontos uniformes, gerados a partir de uma base B formada por vetores. Modelos de criptografia que se baseiam em reticulados utilizam dois problemas difíceis para serem resolvidos, encontrar o menor vetor do reticulado e encontrar a distância entre um vetor arbitrário e um vetor no reticulado.
- **Criptografia baseada em equação quadrática multivariante**: Baseados na dificuldade de resolver sistemas de equações não lineares em campos finitos.

Os outros finalistas na categoria de criptografia de chave públicas e mecanismo de encapsulamento de chaves (CRYSTALS-KYBER, NTRU e SABER) todos utilizam algoritmo baseado em reticulados. Além disso, na categoria de assinaturas, os algoritmos CRYSTALS-DILITHIUM e FALCON são também baseados em reticulados, e o Rainbow baseado em multivariantes.

O Classic McEliece, por ter sido o primeiro de seu tipo e bastante antigo, já foi testado inúmeras vezes ao longo das décadas, e mesmo havendo diversos artigos propondo ataques ele se mantém consistente e com um alto padrão de segurança. Todavia, o custo

disso é que as chaves geradas por esse sistema são muito grandes, da ordem de alguns megabytes. Por conta disso também, é considerado um algoritmo de alto custo para gerar chaves. Outros modelos de criptografia baseada em código foram apresentados ao longo dos anos, para tentar reduzir o tamanho das chaves, mas eles se mostraram menos robustos e seguros.

Entrega 3

1. Introdução

O Classic McEliece é um sistema criptográfico de criptografia assimétrica baseado em códigos lineares de correção de erros, que inclui geração de chaves, criptografia e descriptografia de mensagens. Nesse trabalho, vamos apresentar na prática como ele funciona, utilizando a ferramenta libmcleece disponível em sua página no github.

2. `libmcleece`

A libmcleece foi desenvolvida em C++, e implementa uma interface de linha de comando para a implementação original do Classic McEliece feita para sua sumbissão no concurso de algoritmos de criptografia de chave-pública pós-quânticos do NIST, disponível no site oficial do algoritmo [Bernstein].

Para instalar a libmcleece, basta cloná-la do github, instalar as dependências contidas no arquivo `README.md` e executar os comandos:

```
cmake .
make -j4 install
```

O resultado do build estará disponível, por padrão, na basta `dist/`, e conterá um executável `mcleececli`, que será utilizado nos experimentos que serão realizados.

3. Sistema criptográfico

Nessa seção vamos descrever como funciona cada uma das etapas de criptografia do Classic McEliece: a geração de chave, criptografia e descriptografia de mensagens. Para exemplificar, vamos utilizar os personagens tradicionais de exemplos de criptografia Alice e Bob. Primeiro, Bob vai gerar uma chave e Alice vai encriptar uma mensagem utilizando a chave pública de Bob. Dessa maneira, apenas ele será capaz de descriptografar a mensagem.

Para facilitar o entendimento, primeiro serão descritos cada um dos processos e como eles foram implementados na submissão para o NIST, e depois será apresentado um exemplo prático com parâmetros pequenos, para fins didáticos.

3.1. Geração de Chave

Para Bob gerar um par de chaves, os seguintes passos são executados:

1. Primeiro, ele deve gerar um polinômio irredutível mônico aleatório $g(z) \in F_{2^m}[z]$ de grau t, sendo F_{2^m} um campo de extensão binária (um elemento da aritmética de campos finitos).
2. Seleciona um conjunto uniformemente aleatório $\{\alpha_1, \alpha_2, ..., \alpha_n\} \subseteq F_{2^m}$ com todos os seus elementos sendo distintos.
3. Computar uma matriz de tamanho $t \times n$ definida como $\tilde{H} = \{h_{i,j}\}$ sobre F_{2^m}, onde $\{h_{i,j}\} = a_j^{i-1} g(\alpha_i)^{-1}$ para $i = 1, 2, ..., t$ e $j = 1, 2, ..., n$.
4. Substituir cada entrada da matriz $\tilde{H}$ (elementos de F_{2^m}) com vetores de F_2^m (arranjados em colunas) usando o isomorfismo de espaço vetorial entre F_{2^m} e F_{2^m} para chegar a uma matriz $\hat{H}$ de tamanho $mt \times n$.

5. Aplicar eliminação gaussiana em $\hat{H}$ para obter uma matriz sistemática $H = (I_{mt}|T_{mt\times(n-mt)})$ se possível, caso contrário deve retornar para a etapa 1.
6. Gerar uma cadeia uniforme de bits aleatórios de tamanho n nomeada s.
7. **Chave Pública:** T.
8. **Chave Privada:** $\{s, g(z), \alpha_1, \alpha_2, ..., \alpha_n\}$.

Apesar de ser um pouco difícil de entender, pois utiliza notação matemática pesada e aritmética de campos finitos, a ideia da geração é criar, a partir de um código linear de correção de erros, uma matriz geradora para do código selecionado, e transformar essa matriz em um polinômio.

A implementação da geração de chave privada proposta na submissão do McEliece é a seguinte:

```
int genpoly_gen(gf *out, gf *f)
{
  int i, j, k, c;

  gf mat[ SYS_T+1 ][ SYS_T ];
  gf mask, inv, t;

  // fill matrix

  mat[0][0] = 1;

  for (i = 1; i < SYS_T; i++)
    mat[0][i] = 0;

  for (i = 0; i < SYS_T; i++)
    mat[1][i] = f[i];

  for (j = 2; j <= SYS_T; j++)
    GF_mul(mat[j], mat[j-1], f);

  // gaussian

  for (j = 0; j < SYS_T; j++)
  {
    for (k = j + 1; k < SYS_T; k++)
    {
      mask = gf_iszero(mat[ j ][ j ]);

      for (c = j; c < SYS_T + 1; c++)
        mat[ c ][ j ] ^= mat[ c ][ k ] & mask;

    }

    if ( mat[ j ][ j ] == 0 ) // return if not systematic
```

```
    {
      return -1;
    }

    inv = gf_inv(mat[j][j]);

    for (c = j; c < SYS_T + 1; c++)
      mat[ c ][ j ] = gf_mul(mat[ c ][ j ], inv) ;

    for (k = 0; k < SYS_T; k++)
    {
      if (k != j)
      {
        t = mat[ j ][ k ];

        for (c = j; c < SYS_T + 1; c++)
          mat[ c ][ k ] ^= gf_mul(mat[ c ][ j ], t);
      }
    }
  }

  for (i = 0; i < SYS_T; i++)
    out[i] = mat[ SYS_T ][ i ];

  return 0;
}
```

Nessa implementação, é possível perceber o preenchimento da matriz e a eliminação gaussiana. Para realizar esses procedimentos, é utilizada uma estrutura de dados `gf` que implementa funções para aritmética de campo.

A partir da geração da chave privada, a chave pública é gerada da seguinte maneira:

```
int pk_gen(unsigned char * pk, unsigned char * sk,
uint32_t * perm, int16_t * pi)
{
  int i, j, k;
  int row, c;

  uint64_t buf[ 1 << GFBITS ];

  unsigned char mat[ PK_NROWS ][ SYS_N/8 ];
  unsigned char mask;
  unsigned char b;

  gf g[ SYS_T+1 ]; // Goppa polynomial
  gf L[ SYS_N ]; // support
```

```
gf inv[ SYS_N ];

//

g[ SYS_T ] = 1;

for (i = 0; i < SYS_T; i++) { g[i] = load_gf(sk); sk += 2; }

for (i = 0; i < (1 << GFBITS); i++)
{
  buf[i] = perm[i];
  buf[i] <<= 31;
  buf[i] |= i;
}

uint64_sort(buf, 1 << GFBITS);

for (i = 1; i < (1 << GFBITS); i++)
  if ((buf[i-1] >> 31) == (buf[i] >> 31))
    return -1;

for (i = 0; i < (1 << GFBITS); i++) pi[i] = buf[i] & GFMASK;
for (i = 0; i < SYS_N;         i++) L[i] = bitrev(pi[i]);

// filling the matrix

root(inv, g, L);

for (i = 0; i < SYS_N; i++)
  inv[i] = gf_inv(inv[i]);

for (i = 0; i < PK_NROWS; i++)
for (j = 0; j < SYS_N/8; j++)
  mat[i][j] = 0;

for (i = 0; i < SYS_T; i++)
{
  for (j = 0; j < SYS_N; j+=8)
  for (k = 0; k < GFBITS;  k++)
  {
    b  = (inv[j+7] >> k) & 1; b <<= 1;
    b |= (inv[j+6] >> k) & 1; b <<= 1;
    b |= (inv[j+5] >> k) & 1; b <<= 1;
    b |= (inv[j+4] >> k) & 1; b <<= 1;
    b |= (inv[j+3] >> k) & 1; b <<= 1;
    b |= (inv[j+2] >> k) & 1; b <<= 1;
```

```
    b |= (inv[j+1] >> k) & 1; b <<= 1;
    b |= (inv[j+0] >> k) & 1;

    mat[ i*GFBITS + k ][ j/8 ] = b;
  }

  for (j = 0; j < SYS_N; j++)
    inv[j] = gf_mul(inv[j], L[j]);

}

// gaussian elimination

for (i = 0; i < (PK_NROWS + 7) / 8; i++)
for (j = 0; j < 8; j++)
{
  row = i*8 + j;

  if (row >= PK_NROWS)
    break;

  for (k = row + 1; k < PK_NROWS; k++)
  {
    mask = mat[ row ][ i ] ^ mat[ k ][ i ];
    mask >>= j;
    mask &= 1;
    mask = -mask;

    for (c = 0; c < SYS_N/8; c++)
      mat[ row ][ c ] ^= mat[ k ][ c ] & mask;
  }

  if ( ((mat[ row ][ i ] >> j) & 1) == 0 )
  // return if not systematic
  {
    return -1;
  }

  for (k = 0; k < PK_NROWS; k++)
  {
    if (k != row)
    {
      mask = mat[ k ][ i ] >> j;
      mask &= 1;
      mask = -mask;
```

```
        for (c = 0; c < SYS_N/8; c++)
          mat[ k ][ c ] ^= mat[ row ][ c ] & mask;
      }
    }
  }

  for (i = 0; i < PK_NROWS; i++)
    memcpy(pk + i*PK_ROW_BYTES, mat[i] + PK_NROWS/8, PK_ROW_BYTES);

  return 0;
}
```

3.2. Encriptação

Para Alice realizar o processo de encriptação, ela deve seguir as seguintes etapas:

- Gera um vetor aleatório uniforme $e \in F_2^n$ com peso de Hamming t.
- **Codificação Niederreiter:** Usando a chave pública T de Bob, Alice deve primeiro computar a matriz H, e depois o vetor $C_0 = He = (I|T)e$ de comprimento mt em $F_2^m t$.
- Computar $C_1 = H(2, e)$ e gerar o texto cifrado $C = (C_0, C_1)$ de comprimento $mt + 256$ bits.
- Computar uma sessão de chave de 256 bits $K = H(1, e, C)$.

Na implementação de referência, o vetor e é gerado com o seguinte algoritmo abaixo, que gera valores aleatórios continuamente até encontrar um vetor onde não haja repetição, então realiza as operações necessárias.

```
static void gen_e(unsigned char *e)
{
  int i, j, eq;

  uint16_t ind[ SYS_T ];
  unsigned char bytes[ sizeof(ind) ];
  unsigned char mask;
  unsigned char val[ SYS_T ];

  while (1)
  {
    randombytes(bytes, sizeof(bytes));

    for (i = 0; i < SYS_T; i++)
      ind[i] = load_gf(bytes + i*2);

    // check for repetition

    eq = 0;

    for (i = 1; i < SYS_T; i++)
```

```
      for (j = 0; j < i; j++)
        if (ind[i] == ind[j])
          eq = 1;

    if (eq == 0)
      break;
  }

  for (j = 0; j < SYS_T; j++)
    val[j] = 1 << (ind[j] & 7);

  for (i = 0; i < SYS_N/8; i++)
  {
    e[i] = 0;

    for (j = 0; j < SYS_T; j++)
    {
      mask = same_mask(i, (ind[j] >> 3));

      e[i] |= val[j] & mask;
    }
  }
}
```

Na implementação da codificação de Niederriter, é importante notar que esse algoritmo usa uma síndrome como texto cifrado e a mensagem é um padrão de erro. Isso pode ser visto no trecho de código abaixo:

```
static void syndrome(unsigned char *s,
const unsigned char *pk, unsigned char *e)
{
  unsigned char b, row[SYS_N/8];
  const unsigned char *pk_ptr = pk;

  int i, j;

  for (i = 0; i < SYND_BYTES; i++)
    s[i] = 0;

  for (i = 0; i < PK_NROWS; i++)
  {
    for (j = 0; j < SYS_N/8; j++)
      row[j] = 0;

    for (j = 0; j < PK_ROW_BYTES; j++)
      row[ SYS_N/8 - PK_ROW_BYTES + j ] = pk_ptr[j];

    row[i/8] |= 1 << (i%8);
```

```
    b = 0;
    for (j = 0; j < SYS_N/8; j++)
      b ^= row[j] & e[j];

    b ^= b >> 4;
    b ^= b >> 2;
    b ^= b >> 1;
    b &= 1;

    s[ i/8 ] |= (b << (i%8));

    pk_ptr += PK_ROW_BYTES;
  }
}
```

3.3. Decriptação

Para Bob decriptar a mensagem encriptografada enviada por Alice, os seguintes passos devem ser seguidos:

1. Primeiro, ele deve dividir C como (C_0, C_1), com $C_0 \in F_2^{mt}$ e $C_1 \in F_2^{256}$.
2. Atribuir $b \leftarrow 1$.
3. Decodificação:
   (a) `Entrada:` C_0 e a chava privada $\{s, g(z), \alpha_1, \alpha_2, ..., \alpha_n\}$.
   (b) Estender C_0 para $v = (C_0, 0, ..., 0 \in F_2^{mt}$ inderindo nele $n - mt$ zeros.
   (c) Através da decodificação de Niederreiter, achar a palavra-código c no código Goppa definido por $\Gamma = \{g(z), \alpha_1, \alpha_2, ..., \alpha_n\}$.
   (d) Caso a palavra-código c exista, atribuir o vetor $e = v + c$. Se $wt(e) = t$ e $C_0 = He$, retorna e. Caso contrário, retorna $\perp$.
4. Se a decodificação retornou $\perp$, atribuir $e \leftarrow s$ e $b \leftarrow 0$.
5. Computar $C_1' = H(2, e)$, e verificar se $C_1' = C_1$. Se não for, definir $e \leftarrow s$ e $b \leftarrow 0$.
6. Computar a sessão de chave $K' = H(b, e, C)$.

Se não houver falha em nenhuma das etapas durante o processo de decriptação e $C_1' = C_1$, então certamente a sessão de chave K' será idêntica a K. De maneira equivalente, se Bob receber um texto cifrado válido C, ou seja, $C = (C_0, C_1)$ com $C_0 = He$ para algum $e \in F_2^n$ de peso t e $C_1 = H(2, e)$, a decodificação sempre vai resultar no vetor e.

Na implementação de referência, foi implementado a decriptação de Niederreiter com o decodificador de Berlekamp, um algoritmo decodificador de correção de erros. O seguinte trecho de código recebe a chave privada e o texto crifrado, e retorna o vetor de erro e, conforme descrito anteriormente.

```
int decrypt(unsigned char *e,
const unsigned char *sk, const unsigned char *c)
{
  int i, w = 0;
```

```
  uint16_t check;

  unsigned char r[ SYS_N/8 ];

  gf g[ SYS_T+1 ];
  gf L[ SYS_N ];

  gf s[ SYS_T*2 ];
  gf s_cmp[ SYS_T*2 ];
  gf locator[ SYS_T+1 ];
  gf images[ SYS_N ];

  gf t;

  //

  for (i = 0; i < SYND_BYTES; i++)        r[i] = c[i];
  for (i = SYND_BYTES; i < SYS_N/8; i++) r[i] = 0;

  for (i = 0; i < SYS_T; i++)
  { g[i] = load_gf(sk); sk += 2; } g[ SYS_T ] = 1;

  support_gen(L, sk);

  synd(s, g, L, r);

  bm(locator, s);

  root(images, locator, L);

  //

  for (i = 0; i < SYS_N/8; i++)
    e[i] = 0;

  for (i = 0; i < SYS_N; i++)
  {
    t = gf_iszero(images[i]) & 1;

    e[ i/8 ] |= t << (i%8);
    w += t;

  }

#ifdef KAT
  {
```

```
    int k;
    printf("decrypt e: positions");
    for (k = 0;k < SYS_N;++k)
      if (e[k/8] & (1 << (k&7)))
        printf(" %d",k);
    printf("\n");
  }
#endif

  synd(s_cmp, g, L, e);

  //

  check = w;
  check ^= SYS_T;

  for (i = 0; i < SYS_T*2; i++)
    check |= s[i] ^ s_cmp[i];

  check -= 1;
  check >>= 15;

  return check ^ 1;
}
```

4. Aplicação Prática

É possível utilizar essas três funções da implementação de referência através da libmcleece. Primeiro, para gerar uma chave, basta executar o binário gerado com o build com os seguintes parâmetros:

```
mcleececli generate-keypair --key-path=/tmp/key
```

Com isso, a chave será guardada em um arquivo apontado pelo `--key-path`. Com a chave gerada, podemos encriptar qualquer arquivo através da ferramenta de linha de comando:

```
mcleececli encrypt /caminho/arquivo.ext
  --key-path=/tmp/key.pk > encriptado.bin
```

Por fim, qualquer pessoa que possua o arquivo encriptado e a chave privada, pode decriptar o arquivo com o comando:

```
mcleececli decrypt encriptado.bin
  --key-path=/tmp/key.sk > decriptado
```

5. Conclusão

Apesar da definição formal do algoritmo Classic McEliece ser bastante complexa, utilizando aritmética de campos finitos e bastante notação matemática, os princípios empregados são relativamente simples: utilizar a complexidade computacional de decodificar

códigos lineares de correção de erros para encriptar as mensagens. Isso, em conjunto com uma ferramenta que transporta a implementação de referência para uma ferramenta de interface de linha de comando como o libmcleece, torna o uso do Classic McEliece simples e palpável para aplicações práticas.

Referências

[3Blue1Brown] 3Blue1Brown. Hamming codes and error correction. https://www.youtube.com/watch?v=X8jsijhllIA. Acessado: 18/04/2021.

[Bernstein] Bernstein. Classic McEliece website. https://classic.mceliece.org/index.html. Acessado: 06/04/2021.

[Relyea] Relyea, R. Overview of the nist post quantum algorithms. https://www.youtube.com/watch?v=iPijIw-ZDpY. Acessado: 18/04/2021.