Exercício 3 (LESS) - Trabalho Prático 3

Grupo 6:

Ruben Silva - pg57900

Luís Costa - pg55970

imports

In [ ]:
```
import hashlib
import os
import random as py_random
from sage.all import *
from sage.misc.prandom import randint
from sage.misc.randstate import set_random_seed
```

Paramêtros

In [339…
```
# Set up parameters and field
q = 2  # Field size
F = GF(q)
n = 16  # Code length
k = 8   # Code dimension
t = 10  # Number of rounds
s = 2   # Challenge alphabet size (0, 1, ..., s)
omegam = 3   # Number of non-zero challenge entries
```

Funções Auxiliares

In [340…
```
def HASH(data):
    return hashlib.sha256(data).digest()

def serialize_data(matrix):
    # Ensure consistent serialization
    return str(matrix.list()).encode()

def generate_ch(h, t, s, omegam):
    # Use a deterministic approach based on hash
    seed_val = int.from_bytes(h[:4], 'big') % (2**31)  # Use 4 bytes to avoid ov
    set_random_seed(seed_val)

    # Generate exactly omegam non-zero positions
    positions = list(range(t))
    py_random.Random(seed_val).shuffle(positions)
    non_zero_positions = positions[:omegam]

    ch = [0] * t
    for pos in non_zero_positions:
        ch[pos] = (seed_val + pos) % s + 1  # Deterministic value from 1 to s

    return ch
```

```python
def random_monomial_matrix(F, n, seed=None):
    if seed:
        seed_val = int.from_bytes(seed[:4], 'big') % (2**31)
        set_random_seed(seed_val)

    # Generate random permutation
    perm_list = list(range(n))
    if seed:
        py_random.Random(seed_val).shuffle(perm_list)
    else:
        py_random.shuffle(perm_list)

    # Create monomial matrix
    Q = matrix(F, n, n)
    for i in range(n):
        Q[i, perm_list[i]] = 1  # Use 1 for GF(2)

    return Q
```

Funções Auxilaires Seed n Tree

`compute_node_seed`

O que faz: Calcula a semente de um nó específico em uma árvore de sementes.

Como funciona: A partir de uma semente mestra, usa o caminho do nó na árvore (como uma sequência de escolhas esquerda/direita) para derivar uma semente única. Isso organiza a geração de valores aleatórios em estruturas hierárquicas.

In [ ]:
```python
def compute_node_seed(mseed, node):
    if node == 1:
        return mseed

    path = []
    temp = node
    while temp > 1:
        path.append(temp % 2)
        temp = temp // 2

    seed = mseed
    for bit in reversed(path):
        seed = HASH(seed + bytes([bit]))
    return seed
```

`compute_seed_tree_nodes_to_transmit`

O que faz: Decide quais nós da árvore de sementes precisam ser enviados.

Como funciona: Identifica os nós cujas folhas correspondem a desafios com valor zero. Esses nós são suficientes para o receptor reconstruir a árvore, otimizando a transmissão de dados.

In [ ]:
```python
def compute_seed_tree_nodes_to_transmit(t, ch):
    k_tree = ceil(log(t, 2)) if t > 1 else 1
```

```
    m_tree = 2**k_tree
    leaves = list(range(m_tree, m_tree + t))

    # Find which leaves need to be opened (ch[i] != 0)
    open_leaves = [leaves[i] for i in range(t) if ch[i] != 0]

    # For each closed leaf (ch[i] == 0), we need to provide its seed
    closed_leaves = [leaves[i] for i in range(t) if ch[i] == 0]

    nodes_to_send = set()
    for leaf in closed_leaves:
        current = leaf
        # Add the leaf itself to nodes to send
        nodes_to_send.add(current)

    return list(nodes_to_send)
```

Funções Auxiliares compress/decompress

O que faz: Comprime e descomprime respostas do esquema LESS.

Como funciona: Trabalha com conjuntos de informação para reduzir o tamanho das respostas (assinaturas). Não estão ativas no código atual, mas são projetadas para eficiência em implementações futuras.

In [ ]:
```
def compress_response(G_ch, rsp):
    # Compute G_ch * rsp and put it in echelon form
    G_rsp = (G_ch * rsp).echelon_form()

    # Find information set I (pivot columns)
    I = []
    n_rows = G_rsp.nrows()
    n_cols = G_rsp.ncols()
    for r in range(n_rows):
        for c in range(n_cols):
            if G_rsp[r, c] == 1 and all(G_rsp[i, c] == 0 for i in range(r)):
                I.append(c)
                break
        if len(I) == n_rows:
            break

    # Extract permutation from rsp (π such that rsp[i, π(i)] = 1)
    n = rsp.nrows()
    perm = [next(j for j in range(n) if rsp[i, j] == 1) for i in range(n)]

    # Compute inverse permutation (π^{-1})
    inv_perm = [0] * n
    for i, p in enumerate(perm):
        inv_perm[p] = i

    # J = π^{-1}(I)
    J = [inv_perm[c] for c in I]

    # Scales are all 1 in GF(2)
    compressed_scales = [1] * len(I)

    return I, J, compressed_scales
```

```
def decompress_response(G_ch, I, J, scales):
    n = G_ch.ncols()
    rsp = matrix(GF(2), n, n)

    # Create the permutation matrix
    for j in range(n):
        if j in J:
            # Find corresponding I index
            j_idx = J.index(j)
            rsp[j, I[j_idx]] = scales[j_idx]  # = 1 in GF(2)
        else:
            # Identity for other positions
            rsp[j, j] = 1
    return rsp
```

LESS

`keygen()`

A função gera o par de chaves pública e secreta para o esquema LESS:

1. Define uma matriz identidade $\mathbf{I}_k$ de tamanho $k \times k$ sobre o campo finito $\mathbb{F}$;
2. Gera uma matriz aleatória $\mathbf{M}$ de tamanho $k \times (n-k)$ sobre $\mathbb{F}$;
3. Constrói a matriz geradora $\mathbf{G}_0 = [\mathbf{I}_k | \mathbf{M}]$, que é uma matriz $k \times n$ em forma sistemática;
4. Gera s matrizes monomiais aleatórias $\mathbf{Q}_{\text{invs}} = [\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_s]$, onde cada $\mathbf{Q}_i$ é uma matriz $n \times n$ invertível sobre $\mathbb{F}$, com exatamente um 1 por linha e coluna;
5. Computa as matrizes públicas $\mathbf{G}_s = [\mathbf{G}_1, \mathbf{G}_2, \ldots, \mathbf{G}_s]$, onde cada $\mathbf{G}_i = (\mathbf{G}_0 \cdot \mathbf{Q}_i^{-1})$ é transformada para sua forma escalonada;
6. Define a chave secreta como $sk = (\mathbf{G}0, \mathbf{Q}\text{invs})$ e a chave pública como $pk = (\mathbf{G}_0, \mathbf{G}_s)$;
7. Retorna o par (sk, pk).

In [343...

```
def keygen():
    I_k = identity_matrix(F, k)
    M = random_matrix(F, k, n - k)
    G0 = I_k.augment(M)

    # Generate s different monomial matrices
    Q_invs = []
    for i in range(s):
        Q_inv = random_monomial_matrix(F, n)
        Q_invs.append(Q_inv)

    # Compute public matrices G_s = G0 * Q_inv in echelon form
    G_s = []
    for Q_inv in Q_invs:
        G_i = (G0 * Q_inv).echelon_form()
        G_s.append(G_i)

    sk = (G0, Q_invs)
    pk = (G0, G_s)
    return sk, pk
```

`sign`

A função gera uma assinatura para uma mensagem M usando a chave secreta sk:

1. Desempacota a chave secreta sk em $\mathbf{G}0$ (matriz geradora) e $\mathbf{Q}\mathrm{invs}$ (lista de matrizes monomiais secretas);
2. Gera uma semente mestra $mseed$ como uma sequência aleatória de 32 bytes;
3. Constrói uma árvore de sementes com t folhas, onde cada folha i tem uma semente derivada $seed_i = compute\_node\_seed(mseed, i)$;
4. Gera t matrizes monomiais aleatórias $\mathbf{Qbar}_s = [\mathbf{Qbar}_1, \mathbf{Qbar}_2, \ldots, \mathbf{Qbar}_t]$, cada uma criada com `random_monomial_matrix` usando a semente da respectiva folha;
5. Calcula os compromissos $\mathbf{cmts} = [cmt_1, cmt_2, \ldots, cmt_t]$, onde cada $cmt_i = H(serialize\_data((\mathbf{G}_0 \cdot \mathbf{Qbar}i)\mathrm{echelon}))$, sendo H a função hash SHA-256;
6. Computa o hash do desafio $h = H(H(M)|cmt_1|cmt_2|\ldots|cmt_t)$, concatenando o hash da mensagem com os compromissos;
7. Gera o vetor de desafio $\mathbf{ch} = [ch_1, ch_2, \ldots, ch_t]$ com a função `generate_ch` (h, t, s, ω_m), contendo exatamente ω_m entradas não-zero entre 1 e s;
8. Identifica os nós da árvore de sementes a serem transmitidos com `compute_seed_tree_nodes_to_transmit` $(t, \mathbf{ch})$, selecionando as folhas onde $ch_i = 0$, e empacota os pares $(node, seed)$ correspondentes em seed_tree_data;
9. Computa as respostas $\mathbf{responses} = [resp_1, resp_2, \ldots, resp_t]$, onde:
   - Se $ch_i = 0$, $resp_i = \mathrm{None}$ (não é necessário revelar a resposta);
   - Se $ch_i > 0$, $resp_i = \mathbf{Q}_{\mathrm{invs}}[ch_i - 1]^{-1} \cdot \mathbf{Qbar}_i$, que é a matriz que satisfaz a relação do desafio;
10. Empacota a assinatura como $(\mathbf{ch}, seed\_tree\_data, \mathbf{responses})$ e a retorna.

In [344...

```
def sign(message, sk):
    G0, Q_invs = sk
```

```
    # Generate master seed
    mseed = os.urandom(32)

    # Set up seed tree
    k_tree = ceil(log(t, 2)) if t > 1 else 1
    m_tree = 2**k_tree
    leaves = list(range(m_tree, m_tree + t))

    # Generate commitment matrices
    Qbar_s = []
    for i in range(t):
        leaf_seed = compute_node_seed(mseed, leaves[i])
        Qbar_i = random_monomial_matrix(F, n, seed=leaf_seed)
        Qbar_s.append(Qbar_i)

    # Compute commitments
    cmts = []
    for Qbar in Qbar_s:
        commitment_matrix = (G0 * Qbar).echelon_form()
        cmt = HASH(serialize_data(commitment_matrix))
        cmts.append(cmt)

    # Generate challenge
    hash_input = HASH(message.encode()) + b''.join(cmts)
    h = HASH(hash_input)
    ch = generate_ch(h, t, s, omegam)

    # Determine which seeds to send
    seed_nodes = compute_seed_tree_nodes_to_transmit(t, ch)
    seed_tree_data = []
    for node in seed_nodes:
        seed_tree_data.append((node, compute_node_seed(mseed, node)))

    # Generate responses for non-zero challenges
    responses = []
    for i in range(t):
        if ch[i] == 0:
            responses.append(None)
        else:
            # Response is Q_{ch[i]}^{-1} * Qbar_i
            Q_ch = Q_invs[ch[i] - 1]  # ch[i] is 1-indexed
            response = Q_ch.inverse() * Qbar_s[i]
            responses.append(response)

    return (ch, seed_tree_data, responses)
```

`verify`

A função verifica a validade da assinatura para uma mensagem M usando a chave pública pk:

1. Desempacota a chave pública pk em $\mathbf{G}_0$ e $\mathbf{G}_s$, e a assinatura sig em $\mathbf{ch}$, seed_tree_data e $\mathbf{responses}$;
2. Reconstrói os compromissos $\mathbf{cmts} = [cmt_1, cmt_2, \ldots, cmt_t]$ para cada rodada i:
   - Se $ch_i = 0$, obtém a semente da folha i a partir de $seed_tree_data$, gera $\mathbf{Qbar}_i$ com `random_monomial_matrix`, e calcula

$cmt_i = H(serialize\_data((\mathbf{G}_0 \cdot \mathbf{Qbar}i)\text{echelon}))$;

- $ch_i > 0$, usa a resposta $resp_i$ da assinatura e a matriz pública $\mathbf{G}ch_i - 1$ para calcular $cmt_i = H(serialize\_data((\mathbf{G}ch_i - 1 \cdot resp_i)_{\text{echelon}}))$;

3. Computa o hash $h' = H(H(M)|cmt_1|cmt_2|\ldots|cmt_t)$ com os compromissos reconstruídos;
4. Gera o vetor de desafio reconstruído $\mathbf{ch}' = generate\_ch(h', t, s, \omega_m)$;
5. Verifica se $\mathbf{ch} = \mathbf{ch}'$, retornando True se forem iguais (assinatura válida) ou False caso contrário.

In [345…

```
def verify(message, sig, pk):
    G0, G_s = pk
    ch, seed_tree_data, responses = sig

    # Reconstruct seed tree
    k_tree = ceil(log(t, 2)) if t > 1 else 1
    m_tree = 2**k_tree
    leaves = list(range(m_tree, m_tree + t))

    # Create seed lookup
    known_seeds = {node: seed for node, seed in seed_tree_data}

    # Reconstruct commitments
    cmts = []
    for i in range(t):
        if ch[i] == 0:
            # Use provided seed to reconstruct commitment
            if leaves[i] in known_seeds:
                leaf_seed = known_seeds[leaves[i]]
            else:
                # Try to compute from parent
                leaf_seed = compute_node_seed(known_seeds.get(1, b''), leaves[i]

            Qbar_i = random_monomial_matrix(F, n, seed=leaf_seed)
            commitment_matrix = (G0 * Qbar_i).echelon_form()
        else:
            # Use response to reconstruct commitment
            response = responses[i]
            G_ch = G_s[ch[i] - 1]  # ch[i] is 1-indexed
            commitment_matrix = (G_ch * response).echelon_form()

        cmt = HASH(serialize_data(commitment_matrix))
        cmts.append(cmt)

    # Verify challenge
    hash_input = HASH(message.encode()) + b''.join(cmts)
    h_prime = HASH(hash_input)
    ch_prime = generate_ch(h_prime, t, s, omegam)

    return ch == ch_prime
```

Run

In [346…

```
print("======= LESS ======= ")

sk, pk = keygen()
```

```
print("\nSecret Key\n", sk)
print("\nPublic Key\n", pk)

message = "Secure transaction #123"
print("\n\n======= SIGNING =======:")
sig = sign(message, sk)
print("\nSignature:\n", sig)
is_valid = verify(message, sig, pk)
print("\n\n======= Verify =======:")
print(f"\nSignature valid: {is_valid}")
```

```
======= LESS =======

Secret Key
 ([1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1]
[0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1]
[0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 0]
[0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0]
[0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 1]
[0 0 0 0 0 1 0 0 1 0 1 1 0 0 1 0]
[0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 1]
[0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0], [[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0], [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]])

Public Key
 ([1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1]
[0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1]
[0 0 1 0 0 0 0 0 0 0 0 1 0 1 1 0]
[0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0]
[0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 1]
[0 0 0 0 0 1 0 0 1 0 1 1 0 0 1 0]
[0 0 0 0 0 0 1 0 1 1 0 1 0 0 1 1]
[0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0], [[1 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0]
[0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 0]
[0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 1]
[0 0 0 1 0 0 0 0 0 0 1 0 0 1 1 1]
[0 0 0 0 1 0 0 1 0 1 0 1 0 1 1 1]
[0 0 0 0 0 1 0 0 0 0 1 0 1 1 1 1]
[0 0 0 0 0 0 1 0 0 0 1 1 0 1 1 0]
[0 0 0 0 0 0 0 0 1 1 1 1 0 0 1 0], [1 0 0 0 0 0 0 1 1 0 0 1 1 1 1 0]
[0 1 0 0 0 0 0 1 1 0 0 0 1 1 0 1]
[0 0 1 0 0 0 0 1 1 0 0 0 0 1 0 0]
```

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


======= SIGNING =======:

Signature:
 ([1, 0, 0, 0, 0, 0, 0, 0, 1, 2], [(17, b"\x87 M'\xadx\xa9v\xfa\xb7\xcb\x03\xea\x
dfZ\x87\x1c\xd9\x9bh<\xa3C)N\xef`S5\xe5\x07\xda"), (18, b'r\x13u\xb5z\x8d@\xc9{\x
ceQ$JdX\xeeY\xcau\x1dO\xa3\xf5c\x01\x9eC/\x16\x1aS\x89'), (19, b'\xbf\xe9\xf4K\xf
7\xda\xc1\xf1\xd5E\x9c\x02PT:w1uT\xe3\xfbc\xefE\x85\xfe\xc4\xb5\x05\xc6\x98\xb
9'), (20, b'h&\xecS\xd6u\x85\xa81\xdaO\xce\xab\x85\xb5_q\xbb]d\x0bk\x91\x9e#\xf4H
\xb5f\x96\xc6\xb5'), (21, b'\xda0@v\xb5I\xac\xc4\xc5\xc5\xd7\xc6S\xcb\x1f\x1bmI|
\xe1\xdcq\x01\xb6\x89\xd0"NS-\x1b\xc9'), (22, b'l<-\xe5 +\xc2\x18\xf39$\x1b"\xfaD
\x83o\xc7\t)\xa1\xca\x8cm\xab\x18\x05m\xae\xd3\x8f\xa0'), (23, b'\x03O\xe3\x13\xa
6\x19[_p\x06\xfc!;$\xa5\xa5\xa4\xd2\xa7x\x9e\x99ej\x01\x80\xe9\x9d\xca\xf8\xdb\x0
3')], [[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0], None, None, None, None, None, None, None, [0 0
0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0], [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
```

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


======= Verify =======:

Signature valid: True
```