Enunciado

1. Pretende-se um protocolo ZK baseados na computação sobre circuitos que usem “oblivious transfer” . Para tal
   A. Implemente um algoritmo que, a partir de uma “seed” $\in \{0,1\}^{\lambda}$ aleatoriamente gerada e de um XOF, construa um circuito booleano $n \times 1$ de dimensão poly(n).
   B. Implemente um dos seguintes protocolos com este circuito
      a. O protocolo o protocolo ZK não interactivo de dois passos baseado no modelo “MPC-in-the-Head” com “Oblivious Transfer” (MPCitH-OT) (ver a última secção do Capítulo 6c: Computação Cooperativa).
      b. O protocolo de conhecimento zero com “garbled circuits” e “oblivious transfer” (ZK-GC-OT), ver última secção do Capítulo 6e: “Garbled Circuits” .

Definição de circuito

In [ ]:

```
import hashlib
import os
from collections import OrderedDict
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

class XOF:
    def __init__(self, seed):
        self.shake = hashlib.shake_256()
        self.shake.update(seed)

    def read(self, length):
        return self.shake.digest(length)

class Wire:
    def __init__(self, wire_id):
        self.id = wire_id
        self.token0 = None
        self.token1 = None

class Gate:
    def __init__(self, gate_type, in1, in2, out):
        self.type = gate_type  # 0 para XOR, 1 para AND
        self.in1 = in1 # Wire de entrada 1
        self.in2 = in2 # Wire de entrada 2
        self.out = out # Wire de saida
        self.garbled_table = None

class BooleanCircuit:
    def __init__(self, n_inputs, n_outputs):
        self.n_inputs = n_inputs
        self.n_outputs = n_outputs
        self.wires = OrderedDict()
        self.gates = []
        self.input_wires = []
        self.output_wires = []

    def add_wire(self, wire_id):
        if wire_id not in self.wires:
            self.wires[wire_id] = Wire(wire_id)
        return self.wires[wire_id]

    def add_gate(self, gate_type, in1, in2, out):
        w_in1 = self.add_wire(in1)
        w_in2 = self.add_wire(in2)
        w_out = self.add_wire(out)
        gate = Gate(gate_type, w_in1, w_in2, w_out)
        self.gates.append(gate)
        return gate

    def set_input_wires(self, input_wires):
        self.input_wires = [self.wires[w] for w in input_wires]

    def set_output_wires(self, output_wires):
        self.output_wires = [self.wires[w] for w in output_wires]

    def topological_sort(self):
        in_degree = {wire.id: 0 for wire in self.wires.values()}
        graph = {wire.id: [] for wire in self.wires.values()}

        # Construir o grafo de dependências e calcular graus de entrada
```

```
        for gate in self.gates:
            out_wire = gate.out.id
            in_wire1 = gate.in1.id
            in_wire2 = gate.in2.id

            # Adicionar arestas do input para o output
            if out_wire not in graph[in_wire1]:
                graph[in_wire1].append(out_wire)
                in_degree[out_wire] += 1
            if out_wire not in graph[in_wire2]:
                graph[in_wire2].append(out_wire)
                in_degree[out_wire] += 1

        # Encontrar wires com grau de entrada zero (inputs iniciais)
        queue = [wire_id for wire_id, degree in in_degree.items() if degree == 0]

        # Ordem topológica das portas
        topo_order = []

        # Mapeamento de wire para as portas que o produzem
        wire_to_gate = {}
        for gate in self.gates:
            wire_to_gate[gate.out.id] = gate

        while queue:
            wire_id = queue.pop(0)

            # Se este wire é saída de uma porta, adicionar a porta à ordem
            if wire_id in wire_to_gate:
                gate = wire_to_gate[wire_id]
                topo_order.append(gate)

            # Reduzir grau de entrada dos vizinhos
            for neighbor in graph[wire_id]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    queue.append(neighbor)

        # Verificar se todos os wires foram processados (grafo acíclico)
        if len(topo_order) != len(self.gates):
            raise ValueError("O circuito contém ciclos e não pode ser ordenado topologicamente")

        return topo_order

    def evaluate(self, inputs):
        if len(inputs) != self.n_inputs:
            raise ValueError("Numero de inputs incorreto")

        for wire in self.wires.values():
            wire.value = None

        for i in range(self.n_inputs):
            self.input_wires[i].value = inputs[i]

        for gate in self.topological_sort():
            if gate.in1.value is None or gate.in2.value is None:
                raise ValueError("Gate de input não definido")

            if gate.type == 0:  # XOR
                gate.out.value = gate.in1.value ^ gate.in2.value
            elif gate.type == 1:  # AND
                gate.out.value = gate.in1.value & gate.in2.value
            else:
                raise ValueError("Tipo de gate desconhecido")

        outputs = [wire.value for wire in self.output_wires]
        return outputs
```

Garbled Circuit

Para concretizar este protocolo o esquema necessita de

- Um par de funções (E, D) , “encode” e “decode”, em que ambos as componentes são funções parciais e determinísticas. Cada uma destas funções recebe uma descrição completa das operações que deve executar; essas descrições formam um par (e, d) e pretende-se que seja $x' = \mathrm{E}(e, x) \quad \text{e} \quad y = \mathrm{D}(d, y')$
- Um algoritmo probabilístico Garb que, sob “input” de um parâmetro de segurança κ e da descrição f produz o triplo $\langle f', e, d \rangle$.

In [ ]:

```
# Funções de criptografia AES
def aes_encrypt_block(key: bytes, data: bytes) -> bytes:
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())
    encryptor = cipher.encryptor()
    return encryptor.update(data) + encryptor.finalize()

def aes_decrypt_block(key: bytes, data: bytes) -> bytes:
    cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())
    decryptor = cipher.decryptor()
    return decryptor.update(data) + decryptor.finalize()
```

Para cifrar os circuito, os seus "inputs" e "outputs" vamos apresentar o esquema que consta do artigo referido no início desta secção e que se baseia no uso de "twekable double-key block cipher" . Tomamos como ponto de partida uma cifra simétrica por blocos em modo ECB. Tipicamente usa-se a cifra AES128 agindo sobre mensagens de tamanho $\kappa = 128$ com chaves do mesmo tamanho e com "tweaks" w de tamanho τ.
A função de cifra usa duas chaves k_o, k_1 com κ bits de comprimento e um "tweak" r de tamanho $\tau \leq \kappa$. Para cada par de chaves k_0, k_1 e "tweak" r a cifra é definida pela permutação $F_r(k_0, k_1) : x \mapsto \text{AES}(k_0, (r \,\|\, \text{nounce}) \oplus \text{AES}(k_1, x))$ em que nounce é uma "string" constante de comprimento $(\kappa - \tau)$. Como $F_r(k_0, k_1)$ é uma permutação tem uma função inversa $F_r^{-1}(k_0, k_1)$. No caso particular em que o "tweak" é substituído por uma constante, representamos as permutações simplesmente por $F(k_0, k_1)$ e $F^{-1}(k_0, k_1)$.

In [ ]:

```
def F(self, k0, k1, tweak, x):

        if not isinstance(tweak, bytes):
            tweak = tweak.encode() if isinstance(tweak, str) else bytes([tweak])

        tweak_padded = tweak.ljust(self.security_param//8, b'\x00')
        self.validate_bytes(k0, "k0")
        self.validate_bytes(k1, "k1")
        self.validate_bytes(tweak_padded, "tweak")
        self.validate_bytes(x, "x")

        intermediate = aes_encrypt_block(k1, x)

        # XOR com o paded tweak
        intermediate = bytes([a ^^ b for a, b in zip(intermediate, tweak_padded)])

        return aes_encrypt_block(k0, intermediate)

def F_inv(self, k0, k1, tweak, y):
        if not isinstance(tweak, bytes):
            tweak = tweak.encode() if isinstance(tweak, str) else bytes([tweak])

        tweak_padded = tweak.ljust(self.security_param//8, b'\x00')
        self.validate_bytes(k0, "k0")
        self.validate_bytes(k1, "k1")
        self.validate_bytes(tweak_padded, "tweak")
        self.validate_bytes(y, "y")

        intermediate = aes_decrypt_block(k0, y)

        # XOR com o paded tweak
        intermediate = bytes([a ^^ b for a, b in zip(intermediate, tweak_padded)])

        # Second AES decryption
        return aes_decrypt_block(k1, intermediate)
```

A técnica usada para ofuscar o circuito vai usa "tokens". Tokens são vetores de bits $X \in \{0,1\}^\kappa$ aos quais associamos uma função $\text{sig} : \{0,1\}^\kappa \to \{0,1\}$ designada por assinatura ou tipo da string. Esta assinatura pode ser o bit numa posição particular da palavra (e.g. o bit menos significativo), mas pode ser também uma paridade. Para ofuscar o circuito , $\mathcal{C} = \langle \mathcal{W}, \mathcal{G} \rangle$, associa-se a cada "wire" $w \in \mathcal{W}$ e a cada valor possível do "wire" $v \in \{0,1\}$, um "token" representado por w^v sujeitos à restrição de, para todo $\alpha \in \mathcal{W}$ e todo $v \in \{0,1\}$, verificar-se $\text{sig}(w_\alpha^{1+v}) = 1 + \text{sig}(w_\alpha^v)$

In [ ]:

```
def generate_tokens(self, wire, seed):
    xof0 = XOF(seed + b'token0' + str(wire.id).encode())
    xof1 = XOF(seed + b'token1' + str(wire.id).encode())

    token0 = xof0.read(self.security_param // 8)
    token1 = xof1.read(self.security_param // 8)

    if (token0[-1] & 1) == (token1[-1] & 1):
        token1 = token1[:-1] + bytes([token1[-1] ^ 1])

    if token0 == token1:
        token1 = bytes([token1[0] ^ 1]) + token1[1:]

    wire.token0 = token0
    wire.token1 = token1
    return (token0, token1)

def sig(self, token):
        """Signature function (LSB)"""
        return token[-1] & 1
```

O algoritmo “encode” $x' \leftarrow \mathsf{E}(\mathsf{e}, x)$, com $x \in \{0,1\}^n$, constrói um vetor com n “tokens” selecionando, de entre os pares $\{(w_\alpha^0, w_\alpha^1)\}_{\alpha \in \mathcal{I}}$ que formam e , os “tokens” $\{w_\alpha^{x_\alpha}\}_{\alpha \in \mathcal{I}}$

O algoritmo de “decode” $y \leftarrow \mathsf{D}(\mathsf{d}, y')$ converte um vetor de “tokens” $y' = \{y'_\alpha\}_{\alpha \in \mathcal{O}}$ num vetor de bits $y \in \{y_\alpha\}_{\alpha \in \mathcal{O}}$ Para isso, para cada $\alpha \in \mathcal{O}$, compara-se a componente y'_α com o par de “tokens” (w_α^0, w_α^1) recuperado de d; sinteticamente isto é, determina-se y_α como o bit $b \in \{0,1\}$ tal que $w_\alpha^b = y'_\alpha$; se não existir tal bit então o algoritmo termina em falha.

In [ ]:

```
def encode(self, e, x):
        if len(x) != len(e):
            raise ValueError("Input length doesn't match encode information")

        return [e[i][0] if x[i] == 0 else e[i][1] for i in range(len(x))]

def decode(self, d, y_prime):
    y = []
    for i in range(len(y_prime)):
        if y_prime[i] == d[i][0]:
            y.append(0)
        elif y_prime[i] == d[i][1]:
            y.append(1)
        else:
            raise ValueError("Invalid token in output")
    return y
```

O algoritmo $\mathsf{eval}'(f', x')$ reconstrói os “tokens” w_α que representam os valores no diferentes valores em todos os “wires” do circuito. Os “tokens” nos “input wires” já estão calculados em x'; todos os restantes “tokens” denotam valores do “output” de uma gate $g' \in \mathcal{G}'$.
Decompondo a “gate” nas suas componentes tem-se $g' = (T, \alpha, \beta, \gamma)$ em que, percorrendo $\mathcal{G}$ na ordenação topológica, parte-se do princípio que já são conhecidos os “tokens” w_α e w_β e pretende-se calcular o “token” w_γ. Recordando que a tabela T contém a cifra do “token” w_γ para diferentes combinações das assinaturas de w_α e w_β precisamos começar por identificar essas assinaturas $a \leftarrow \mathsf{str}(w_\alpha)$; $b \leftarrow \mathsf{str}(w_\beta)$ Em seguida é preciso reconstruir o “tweak” . Sendo g' uma ofuscação de $g = (t, \alpha, \beta, \gamma)$ então se $\mathsf{ord}(g) = \mathsf{ord}(g')$. Portanto o “tweak” r pode ser reconstruído como $r \leftarrow \mathsf{ord}(g') \,\|\, a \,\|\, b$ e, finalmente, pode-se decifrar $T_{a,b}$ e calcular $w_\gamma \leftarrow F_r^{-1}(w_\alpha, w_\beta)(T_{a,b})$

In [ ]:

```
def eval_garbled(self, garbled_circuit, x_prime):
        wire_values = {}

        # Verifica se todos os input wires existem
        if len(x_prime) != len(garbled_circuit.input_wires):
            raise ValueError("Número de inputs não corresponde aos wires de entrada")

        # Inicializa os valores dos wires de entrada
        for i, wire in enumerate(garbled_circuit.input_wires):
            wire_values[wire.id] = x_prime[i]

        # Processa cada gate em ordem topológica
        for gate in garbled_circuit.topological_sort():
            # Verifica se os inputs do gate estão disponíveis
            if gate.in1.id not in wire_values:
                raise KeyError(f"Input wire {gate.in1.id} não encontrado")
            if gate.in2.id not in wire_values:
                raise KeyError(f"Input wire {gate.in2.id} não encontrado")

            k0 = wire_values[gate.in1.id]
            k1 = wire_values[gate.in2.id]

            a = self.sig(k0)
            b = self.sig(k1)

            r = self.ord(gate) + bytes([a]) + bytes([b])

            try:
                encrypted_output = gate.garbled_table[(a, b)]
            except KeyError:
                raise KeyError(f"Entrada ({a}, {b}) não encontrada na tabela do gate {gate.in1.id},{gate.in2.id}-
>{gate.out.id}")

            output_token = self.F_inv(k0, k1, r, encrypted_output)
            wire_values[gate.out.id] = output_token

        # Verifica se todos os output wires existem
        for wire in garbled_circuit.output_wires:
            if wire.id not in wire_values:
                raise KeyError(f"Output wire {wire.id} não encontrado")

        return [wire_values[wire.id] for wire in garbled_circuit.output_wires]
```

Código completo GarbledCircuit

In [ ]:

```
class GarbledCircuit:
    def __init__(self, security_param=128):
        self.security_param = security_param

    def validate_bytes(self, value, name):
        if not isinstance(value, bytes):
            raise TypeError(f"{name} must be bytes, got {type(value)}")
        if any(b > 255 or b < 0 for b in value):
            raise ValueError(f"{name} contains invalid byte values")

    def generate_tokens(self, wire, seed):
        xof0 = XOF(seed + b'token0' + str(wire.id).encode())
        xof1 = XOF(seed + b'token1' + str(wire.id).encode())

        token0 = xof0.read(self.security_param // 8)
        token1 = xof1.read(self.security_param // 8)

        if (token0[-1] & 1) == (token1[-1] & 1):
            token1 = token1[:-1] + bytes([token1[-1] ^ 1])

        if token0 == token1:
            token1 = bytes([token1[0] ^ 1]) + token1[1:]

        wire.token0 = token0
        wire.token1 = token1
        return (token0, token1)

    def F(self, k0, k1, tweak, x):

        if not isinstance(tweak, bytes):
            tweak = tweak.encode() if isinstance(tweak, str) else bytes([tweak])

        tweak_padded = tweak.ljust(self.security_param//8, b'\x00')
```

```
        tweak_padded = tweak.ljust(self.security_param//8, b'\x00')
        self.validate_bytes(k0, "k0")
        self.validate_bytes(k1, "k1")
        self.validate_bytes(tweak_padded, "tweak")
        self.validate_bytes(x, "x")

        intermediate = aes_encrypt_block(k1, x)

        # XOR com o paded tweak
        intermediate = bytes([a ^^ b for a, b in zip(intermediate, tweak_padded)])

        return aes_encrypt_block(k0, intermediate)

    def F_inv(self, k0, k1, tweak, y):
        if not isinstance(tweak, bytes):
            tweak = tweak.encode() if isinstance(tweak, str) else bytes([tweak])

        tweak_padded = tweak.ljust(self.security_param//8, b'\x00')
        self.validate_bytes(k0, "k0")
        self.validate_bytes(k1, "k1")
        self.validate_bytes(tweak_padded, "tweak")
        self.validate_bytes(y, "y")

        intermediate = aes_decrypt_block(k0, y)

        # XOR com o paded tweak
        intermediate = bytes([a ^^ b for a, b in zip(intermediate, tweak_padded)])

        # Second AES decryption
        return aes_decrypt_block(k1, intermediate)

    def sig(self, token):
        return token[-1] & 1

    def ord(self, gate):
        index = self.circuit.gates.index(gate)
        return index.to_bytes(4, 'big')

    def garble(self, circuit, seed):
        self.circuit = circuit

        # Gera tokens para cada wire
        for wire in circuit.wires.values():
            self.generate_tokens(wire, seed)

        # cria a tabela de garbled para cada gate
        for gate in circuit.gates:
            T = {}

            for v in [0, 1]:
                for u in [0, 1]:
                    k0 = gate.in1.token0 if v == 0 else gate.in1.token1
                    k1 = gate.in2.token0 if u == 0 else gate.in2.token1

                    a = self.sig(k0)
                    b = self.sig(k1)

                    # Calcula tweak (ord(g) + a + b)
                    r = self.ord(gate) + bytes([a]) + bytes([b])

                    if gate.type == 0:  # XOR
                        val = v ^ u
                    else:  # AND
                        val = v & u

                    output_token = gate.out.token0 if val == 0 else gate.out.token1
                    T[(a, b)] = self.F(k0, k1, r, output_token)

            gate.garbled_table = T

        e = [(wire.token0, wire.token1) for wire in circuit.input_wires]
        d = [(wire.token0, wire.token1) for wire in circuit.output_wires]

        # O circuito garbled é o mesmo circuito, mas com as tabelas de garbled
        garbled_circuit = circuit

        return (garbled_circuit, e, d)

    def encode(self, e, x):
        if len(x) != len(e):
            raise ValueError("Input length doesn't match encode information")

        return [e[i][0] if x[i] == 0 else e[i][1] for i in range(len(x))]
```

```
    def decode(self, d, y_prime):

        y = []
        for i in range(len(y_prime)):
            if y_prime[i] == d[i][0]:
                y.append(0)
            elif y_prime[i] == d[i][1]:
                y.append(1)
            else:
                raise ValueError("Invalid token in output")
        return y

    def eval_garbled(self, garbled_circuit, x_prime):
        wire_values = {}

        # Verifica se todos os input wires existem
        if len(x_prime) != len(garbled_circuit.input_wires):
            raise ValueError("Número de inputs não corresponde aos wires de entrada")

        # Inicializa os valores dos wires de entrada
        for i, wire in enumerate(garbled_circuit.input_wires):
            wire_values[wire.id] = x_prime[i]

        # Processa cada gate em ordem topológica
        for gate in garbled_circuit.topological_sort():
            # Verifica se os inputs do gate estão disponíveis
            if gate.in1.id not in wire_values:
                raise KeyError(f"Input wire {gate.in1.id} não encontrado")
            if gate.in2.id not in wire_values:
                raise KeyError(f"Input wire {gate.in2.id} não encontrado")

            k0 = wire_values[gate.in1.id]
            k1 = wire_values[gate.in2.id]

            a = self.sig(k0)
            b = self.sig(k1)

            r = self.ord(gate) + bytes([a]) + bytes([b])

            try:
                encrypted_output = gate.garbled_table[(a, b)]
            except KeyError:
                raise KeyError(f"Entrada ({a}, {b}) não encontrada na tabela do gate {gate.in1.id},{gate.in2.id}-
>{gate.out.id}")

            output_token = self.F_inv(k0, k1, r, encrypted_output)
            wire_values[gate.out.id] = output_token

        # Verifica se todos os output wires existem
        for wire in garbled_circuit.output_wires:
            if wire.id not in wire_values:
                raise KeyError(f"Output wire {wire.id} não encontrado")

        return [wire_values[wire.id] for wire in garbled_circuit.output_wires]
```

ObliviousTransfer

In [ ]:

```
class ObliviousTransfer:
    # Protocolo simplificado de OT 1-out-of-2
    @staticmethod
    def choose(sid, choice_bit):
        # Parte do receptor - escolhe qual a mensagem que receberá
        return (sid, choice_bit)

    @staticmethod
    def transfer(sid, m0, m1, choice_info):
        # Parte do emissor - envia uma mensagem baseada na escolha
        _, choice_bit = choice_info
        return m0 if choice_bit == 0 else m1
```

ZK-GC-OT

Neste protocolo ZK o verifier actua como sender nos vários protocolos OT que vão ser usados e o prover actua como receiver dos OT's.

Para cada sessão sid , ambos os agentes desenvolvem o seguinte protocolo

1. O prover (actuando como receiver) e o verifier (actuando como sender) iniciam n versões do protocolo $\binom{2}{1}OT$.
   Para todo $i = 1, \cdots, n$, no i-ésimo protocolo OT executam $\mathsf{Choose}(\mathsf{sid}, x_i)$.
2. O verifier/sender:
   A. gera o circuito "garbled" $(f', e, d) \leftarrow \mathsf{Garb}(\kappa, f)$ a partir da função f e do parâmetro de segurança κ .
   B. para cada $i = 1, \cdots, n$ seleciona $(w_i^0, w_i^1) \in e$, e no i-ésimo protocolo OT executa $\mathsf{Transfer}(\mathsf{id}, w_i^0, w_i^1)$. Como resultado $w_i^{x_i}$ é transferido para o prover, via OT. O tuplo $x' = \{w_i^{x_i}\}_{i=1}^n$ forma a versão "garbled" do input que é, desta forma, transferida para o prover.
   C. f' é transferido directamente para prover .
3. O prover recebeu do verifier as versões "garbled" x' , f' . Por isso executa $y' \leftarrow \mathsf{eval}'(f', x_1', \cdots, x_n')$ e envia este resultado para o verifier.
4. O verifier descodifica o valor de y' calculando $y \leftarrow \mathsf{D}(d, y')$; aceita a prova sse $y \overset{?}{=} 1$.

In [ ]:

```
class ZKGCOTProtocol:
    def __init__(self, security_param=128):
        self.security_param = security_param
        self.gc = GarbledCircuit(security_param)
        self.ot = ObliviousTransfer()

    def build_circuit_from_seed(self, seed, n):
        xof = XOF(seed)
        num_gates = n * 3  # 3 gates por input wire
        num_wires = n + num_gates + 1  # inputs + gates + output

        circuit = BooleanCircuit(n, 1)
        wire_ids = [f"w{i}" for i in range(num_wires)]

        for wid in wire_ids:
            circuit.add_wire(wid)

        # Define input e output wires
        circuit.set_input_wires(wire_ids[:n])
        circuit.set_output_wires([wire_ids[-1]])  # Ultimo wire é a saída

        # Construir o circuito
        for i in range(num_gates):
            gate_bytes = xof.read(3)
            gate_type = gate_bytes[0] % 2

            # Qualquer wire anterior pode ser usado como input
            in1_idx = gate_bytes[1] % (n + i)
            in2_idx = gate_bytes[2] % (n + i)

            # O wire de saída é o próximo wire disponível
            out_idx = n + i

            # Nao criar ciclos
            if out_idx <= max(in1_idx, in2_idx):
                in1_idx = max(0, out_idx - 2)
                in2_idx = max(0, out_idx - 1)

            if in1_idx == in2_idx:
                in2_idx = (in2_idx + 1) % (n + i)  # Pega o próximo wire
                if in2_idx == in1_idx:  # Caso extremo (apenas 1 wire disponível)
                    in2_idx = 0  # Usa o primeiro wire como fallback

            circuit.add_gate(gate_type, wire_ids[in1_idx], wire_ids[in2_idx], wire_ids[out_idx])

        # Força o ultimo gate a conectar ao wire de saída
        if num_gates > 0:
            last_gate = circuit.gates[-1]
            if last_gate.out.id != wire_ids[-1]:
                # Reconecta o último gate ao wire de saída
                circuit.add_gate(last_gate.type, last_gate.in1.id, last_gate.in2.id, wire_ids[-1])

        return circuit

    def prover(self, sid, x, f_prime, ot_choices):
        # Receive tokens via OT
        x_prime = []
        for i in range(len(x)):
            ot_choice = self.ot.choose(sid + str(i).encode(), x[i])
            received_token = self.ot.transfer(sid + str(i).encode(),
                                              f_prime[1][i][0], f_prime[1][i][1],
                                              ot_choice)
            x_prime.append(received_token)
```

```
        # Avalia o circuito garbled
        y_prime = self.gc.eval_garbled(f_prime[0], x_prime)

        return y_prime

    def verifier(self, sid, f, x, circuit):
        # Garble o circuito
        seed = os.urandom(self.security_param // 8)
        f_prime = self.gc.garble(circuit, seed)

        # Faz transferência de OT para cada bit de entrada
        ot_choices = []
        for i in range(len(x)):
            ot_choice = self.ot.choose(sid + str(i).encode(), x[i])
            ot_choices.append(ot_choice)

        # Envia o f_prime para o prover
        return f_prime, ot_choices

    def verify_output(self, d, y_prime):
        """Verify the output"""
        y = self.gc.decode(d, y_prime)
        return y[0] == 1
```

Testes

In [ ]:

```
security_param = 128
protocol = ZKGCOTProtocol(security_param)

n = 4  # Numero de wires de entrada
seed = os.urandom(security_param // 8)
print(f"Seed: {seed.hex()}")
circuit = protocol.build_circuit_from_seed(seed, n)

print("Circuito construido com sucesso!")
print(f"Input wires: {[w.id for w in circuit.input_wires]}")
print(f"Output wires: {[w.id for w in circuit.output_wires]}")
print(f"Numero de gates: {len(circuit.gates)}")
```

In [ ]:

```
def find_valid_inputs(circuit, max_inputs=None):
    from itertools import product, islice

    n = circuit.n_inputs
    valid_inputs = []

    all_combinations = product([0, 1], repeat=n)

    if max_inputs is not None:
        all_combinations = islice(all_combinations, max_inputs)

    for input_combination in all_combinations:
        try:
            output = circuit.evaluate(input_combination)
            if output[0] == 1:
                valid_inputs.append(input_combination)
        except Exception as e:
            print(f"Erro ao avaliar {input_combination}: {str(e)}")
            continue

    return valid_inputs

valid_inputs = find_valid_inputs(circuit)
print(f"Combinações válidas encontradas: {len(valid_inputs)}")
for i, inputs in enumerate(valid_inputs):
    print(f"{i+1}. {inputs} → {circuit.evaluate(inputs)}")
```

In [ ]:

```
x = [1, 1, 1, 1]

sid = b"session1"

# Verifier
f_prime, ot_choices = protocol.verifier(sid, None, x, circuit)

# Prover
y_prime = protocol.prover(sid, x, f_prime, ot_choices)

# Verifica o output do verifier
is_valid = protocol.verify_output(f_prime[2], y_prime)

print(f"Prova é valida: {is_valid}")
```

Debugs

In [ ]:

```
try:
    print("\n=== Circuit Structure ===")
    print(f"Input wires: {[w.id for w in circuit.input_wires]}")
    print(f"Output wires: {[w.id for w in circuit.output_wires]}")
    print(f"Total gates: {len(circuit.gates)}")
    print(f"Total wires: {len(circuit.wires)}")

    print("\n=== Token Generation Test ===")
    test_wire = circuit.input_wires[0]
    t0, t1 = protocol.gc.generate_tokens(test_wire, seed)
    print(f"Test tokens for {test_wire.id}:")
    print(f"Token0: {t0.hex()}, LSB={t0[-1] & 1}")
    print(f"Token1: {t1.hex()}, LSB={t1[-1] & 1}")

    print("\n=== Running Protocol ===")
    x = [1, 1, 1, 1]
    sid = b"session1"

    print("Verifier garbling circuit...")
    f_prime, ot_choices = protocol.verifier(sid, None, x, circuit)

    print("\nProver evaluating...")
    y_prime = protocol.prover(sid, x, f_prime, ot_choices)

    print("\nVerifier validating...")
    is_valid = protocol.verify_output(f_prime[2], y_prime)

    print(f"\nFinal Result: Proof is {'valid' if is_valid else 'INVALID'}")

except Exception as e:
    print(f"\n=== ERROR ===")
    print(f"Type: {type(e).__name__}")
    print(f"Message: {str(e)}")
    print("\nDebug Info:")
    print(f"Seed used: {seed.hex()}")

    if 'circuit' in locals():
        print("\nCircuit State:")
        print(f"Gates: {len(circuit.gates)}")
        if len(circuit.gates) > 0:
            last_gate = circuit.gates[-1]
            print(f"Last gate: {last_gate.in1.id},{last_gate.in2.id}->{last_gate.out.id}")

    if 'f_prime' in locals():
        print("\nGarbled Circuit State:")
        print(f"Output tokens: {[t[0].hex()[:8]+'...' for t in f_prime[2]]}")

    import traceback
    print("\nStack Trace:")
    traceback.print_exc()
```