



**UNIVERSIDADE FEDERAL DE SÃO CARLOS**

**DEPARTAMENTO DE COMPUTAÇÃO**

**SEGURANÇA CIBERNÉTICA**

**DOCENTE RESPONSÁVEL:** Paulo Matias

Bruno Nieri Nunes - 820590  
Gustavo Kim Alcantara - 820763  
Guilherme Bartoletti Oliveira - 821881  
Lucas Mantovani - 794040  
Maykon dos Santos Gonçalves - 821653  
Pietro Bernardo Dutra Scaglione - 824375  
Tiago de Paula Evangelista - 824369  
Vinícius Marto da Veiga - 821252

**SÃO CARLOS – SP**  
**2025**

## 1. Introdução

A criptografia de circuitos embaralhados, ou Garbled Circuits, é uma técnica fundamental em criptografia que permite a duas partes (tradicionalmente chamadas de Alice e Bob) computarem uma função em conjunto, sem que nenhuma delas revele sua entrada privada para a outra. Esse conceito é popularmente ilustrado pelo clássico "Dilema dos Milionários de Yao". O objetivo é simples: descobrir quem é o mais rico sem que Alice revele sua riqueza a Bob, e vice-versa.

Para alcançar isso, um circuito booleano é criptografado e "embaralhado". O processo envolve o Garbler (construtor), que cria o circuito embaralhado, e o Evaluator (avaliador), que realiza o cálculo dos resultados. Cada fio (ou "wire") do circuito possui dois "labels" — um para cada possível valor de entrada, como 0 ou 1. Para cada porta lógica do circuito, uma tabela verdade embaralhada é gerada, contendo todas as combinações de labels de entrada e saída.

O cenário ideal de uso funciona da seguinte forma: o Garbler (Alice) envia o circuito embaralhado para o Evaluator (Bob). Bob obtém os labels correspondentes à sua própria entrada de forma segura e recebe os labels de Alice. Com um label para cada entrada, Bob consegue decifrar apenas um único caminho através do circuito para obter o label de saída final. Ele então devolve esse label para Alice, que é a única que sabe se ele representa o valor 0 ou 1.

## 2. Descrição Geral do desafio

O desafio proposto inverte o cenário de colaboração e segurança. Nele, você assume o papel de um atacante, não de um Avaliador honesto. O objetivo principal não é cooperar para calcular a função, mas sim explorar falhas na implementação do circuito para descobrir os labels secretos do servidor. Esses labels correspondem à flag do desafio (CTF).

A arquitetura do desafio é composta por sete arquivos principais:

- **block\_cipher.py**: Implementa a criptografia utilizada pelo [yao.py](#).
- **circuit.json**: Descreve a estrutura do circuito booleano, com quatro entradas e uma saída. A lógica é que a saída só será "verdadeira" se todas as quatro entradas forem "verdadeiras".
- **evaluate\_garbled\_circuit\_example.py**: Um arquivo de exemplo que demonstra como um circuito seria avaliado se as chaves de entrada já fossem conhecidas.
- **generate\_garbled\_circuit.py**: Outro arquivo de exemplo que mostra como o circuito embaralhado foi gerado.
- **obtain\_flag.py**: Este é o script alvo do desafio. Você deve preencher o dicionário de entradas com os labels corretos para obter a flag. O script verifica se as entradas produzem uma saída "verdadeira", calcula um hash SHA512 e executa uma operação XOR para revelar a flag.

- **public\_data.py**: Contém as tabelas embaralhadas (**g\_tables**) das portas 5, 6 e 7, que são a parte pública do circuito a ser analisada.
- **yao.py**: Implementa o protocolo de Garbled Circuits, contendo as classes e funções principais para o Garbler e o Evaluator.

A chave para o ataque reside nas vulnerabilidades da implementação, como o uso de chaves curtas (24 bits), encriptação de um valor conhecido, encriptação determinística e a reutilização de primitivas de criptografia entre as portas. Essas falhas possibilitam um ataque de meet-in-the-middle para recuperar os labels de entrada e, por fim, a flag.

### 3. Teoria por trás

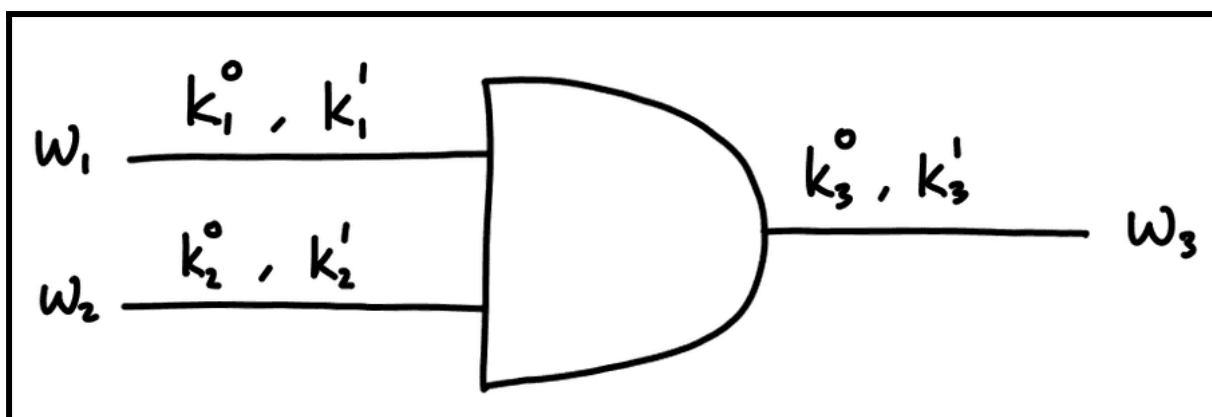
Para entender o desafio, suas vulnerabilidades e como podemos resolvê-lo, inicialmente precisamos entender sua base teórica por trás. Diante disso, vamos compreender o que são os Garbled Circuits(Circuitos embaralhados) e como eles são utilizados no desafio.

A motivação para os Circuitos Embaralhados é melhor ilustrada pelo clássico "Dilema dos Milionários de Yao". O cenário é o seguinte: duas partes (Alice e Bob, por exemplo) desejam calcular uma função baseada em suas informações privadas (por exemplo, "quem é mais rico?") sem revelar essas informações uma à outra.

Para realizar este cálculo, utilizamos de uma circuito booleano tradicional (AND,OR,XOR) e “embaralhamos” e criptografamos ele. O processo envolve alguns atores e componentes:

- Atores:
  - Garbler(“Embaralhador”/construtor): cria e embaralha o circuito
  - Evaluator(Avaliador): quem vai de fato calcular os resultados sobre o circuito
- Componentes:
  - Labels: cada fio do circuito possui dois labels, um para cada entrada(EX: fio A possui labels KA\_0 e KA\_1).
  - Tabelas Verdades Embaralhadas: cada porta lógica do circuito recebe uma dessas tabelas com todas as possibilidades de combinações de labels de entrada e saída

Exemplo de um circuito embaralhado com labels:



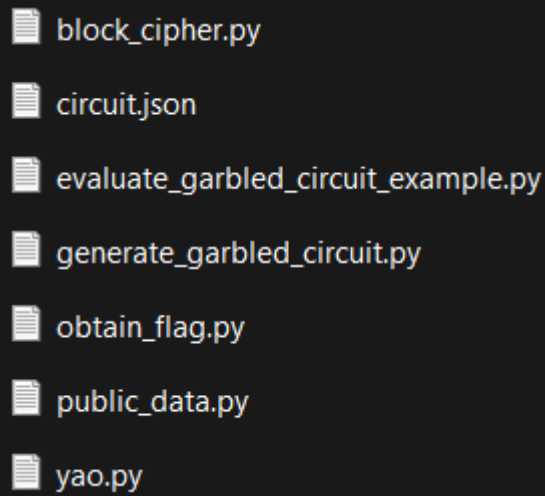
Exemplo de uma tabela verdade embaralhada:

$w_1$	$w_2$	$w_3$
$k_1^0$	$k_2^0$	$E_{k_1^0}(E_{k_2^0}(k_3^0))$
$k_1^0$	$k_2^1$	$E_{k_1^0}(E_{k_2^1}(k_3^0))$
$k_1^1$	$k_2^0$	$E_{k_1^1}(E_{k_2^0}(k_3^0))$
$k_1^1$	$k_2^1$	$E_{k_1^1}(E_{k_2^1}(k_3^1))$

O cenário seguro funciona assim: o Garbler (Alice) envia o circuito embaralhado para o Evaluator (Bob). Bob obtém os rótulos de sua própria entrada de forma segura (via Oblivious Transfer) e recebe de Alice os rótulos da entrada dela. Com um rótulo para cada entrada, Bob só consegue decifrar um único caminho através do circuito, obtendo o rótulo de saída final. Ele devolve esse rótulo para Alice, que é a única que sabe se ele significa 0 ou 1.

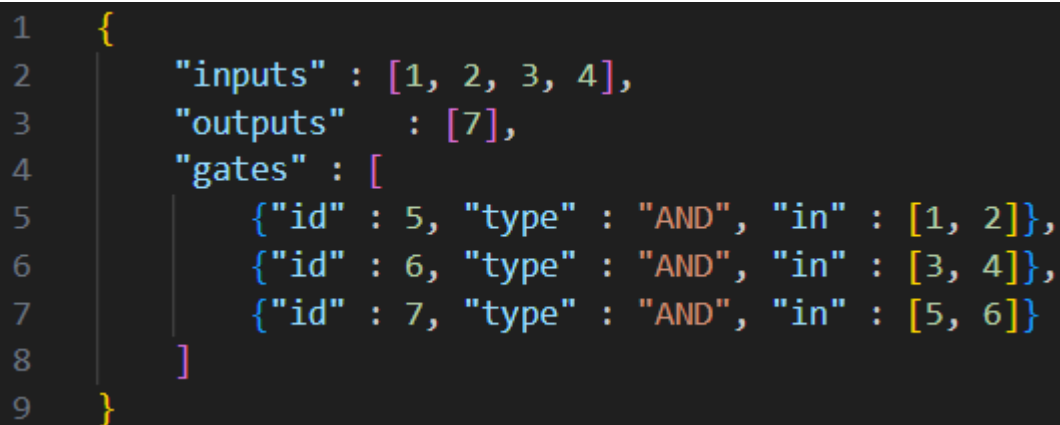
Em relação ao desafio proposto, os papéis são os mesmos: nós somos o Avaliador e o servidor é o Embaralhador. Contudo, o objetivo se inverte fundamentalmente. Diferente do cenário seguro, onde as partes cooperam para proteger seus dados, aqui o nosso papel não é o de um Avaliador honesto. Somos um atacante, e nossa missão não é calcular a função, mas sim explorar falhas na implementação do circuito para descobrir os rótulos secretos do servidor, rótulos que, em um cenário ideal, ele jamais nos forneceria e que, no desafio, correspondem à flag(CTF).

## 4. Investigando código fonte



block\_cipher.py  
circuit.json  
evaluate\_garbled\_circuit\_example.py  
generate\_garbled\_circuit.py  
obtain\_flag.py  
public\_data.py  
yao.py

O problema é constituído por 7 arquivos:



```
1  {
2      "inputs" : [1, 2, 3, 4],
3      "outputs" : [7],
4      "gates" : [
5          {"id" : 5, "type" : "AND", "in" : [1, 2]},
6          {"id" : 6, "type" : "AND", "in" : [3, 4]},
7          {"id" : 7, "type" : "AND", "in" : [5, 6]}
8      ]
9  }
```

- circuit.json: descreve a estrutura do circuito booleano, com 4 entradas e 1 saída. A lógica é:
  - Porta 5: Entrada 1 AND Entrada 2
  - Porta 6: Entrada 3 AND Entrada 4
  - Porta 7 (Saída): Porta 5 AND Porta 6
  - Ou seja, Saída = Entrada 1 AND Entrada 2 AND Entrada 3 AND Entrada 4, todas as entradas devem ser verdadeiras

```

1  from block_cipher import encrypt, decrypt
2  from random import shuffle, randrange
3
4
5  > def generate_random_label(): ...
7
8  > def garble_label(key0, key1, key2): ...
16
17 > def evaluate_gate(garbled_table, key0, key1): ...
31
32 > def evaluate_circuit(circuit, g_tables, inputs): ...
68
69 > class GarbledGate: ...
116
117 > class GarbledCircuit: ...

```

- yao.py: implementação do protocolo de Garbled Circuits

```

117  class GarbledCircuit:
118      """
119      A representation of a garbled circuit.
120      Keyword arguments:
121      circuit -- dict containing circuit spec
122      """
123
124  > def __init__(self, circuit): ...
140
141  > def _gen_keys(self): ...
148
149  > def _gen_garbled_tables(self): ...
154
155  > def get_garbled_tables(self): ...
158
159  > def get_keys(self): ...

```

- GarbledCircuit é a classe que o Garbler usaria para gerar as chaves e as tabelas embaralhadas

```

32  ✓ def evaluate_circuit(circuit, g_tables, inputs):
33  ✓     """
34      Evaluate yao circuit with given inputs.
35
36      Keyword arguments:
37      circuit    -- dict containing circuit spec
38      g_tables   -- garbled tables of yao circuit
39      inputs     -- dict mapping wires to labels
40
41      Returns:
42      evaluation -- a dict mapping output wires to the result labels
43      """
44      gates      = circuit["gates"] # dict containing circuit gates
45      wire_outputs = circuit["outputs"] # list of output wires
46      wire_inputs  = {}              # dict containing Alice and Bob inputs
47      evaluation   = {}              # dict containing result of evaluation
48
49      wire_inputs.update(inputs)
50
51      # Iterate over all gates
52  >   for gate in sorted(gates, key=lambda g: g["id"]): ...
62
63      # After all gates have been evaluated, we populate the dict of results
64  >   for out in wire_outputs: ...
66
67      return evaluation

```

- evaluate\_circuit é a função principal do Evaluator, recebendo o circuito, as tabelas (g\_tables) e os rótulos de entrada para calcular o rótulo de saída

```

1  SBoxes = [[15, 1, 7, 0, 9, 6, 2, 14, 11, 8, 5, 3, 12, 13
2
3  SInvBoxes = [[3, 1, 6, 11, 14, 10, 5, 2, 9, 4, 15, 8, 12
4
5  > def S(block, SBoxes): ...
11
12  PBox = [13, 3, 15, 23, 6, 5, 22, 21, 19, 1, 18, 17, 20,
13  PInvBox = [21, 9, 17, 1, 23, 5, 4, 14, 15, 19, 13, 22, 1
14
15  > def permute(block, pbox): ...
21
22  > def encrypt_data(block, key): ...
32
33  > def decrypt_data(block, key): ...
43
44  > def encrypt(data, key1, key2): ...
48
49  > def decrypt(data, key1, key2): ...

```

- block\_cipher.py: implementação da criptografia (rede de substituição-permutação) usado pelo yao.py

```

1  g_tables = {5: [(5737111, 2983937),
2  (15406556, 16284948),
3  (14172222, 14132908),
4  (4000971, 16383744)],
5  6: [(8204186, 1546264),
6  (229766, 3208405),
7  (9550202, 13483954),
8  (13257058, 5195482)],
9  7: [(1658768, 11512735),
10  (1023507, 9621913),
11  (7805976, 1206540),
12  (2769364, 9224729)]}

```

- public\_data.py: contém as tabelas embaralhadas (g\_tables) para as portas 5, 6 e 7. É a parte pública do circuito que deve ser analisada



```

6  import hashlib
7  import json
8  from yao import evaluate_circuit
9  from public_data import g_tables
10 from private_data import keys, flag
11
12 def xor(A, B):
13     return bytes(a ^ b for a, b in zip(A, B))
14
15 circuit_filename = "circuit.json"
16 with open(circuit_filename) as json_file:
17     circuit = json.load(json_file)
18
19 # ?????????????????
20 inputs = { 1: ?????????????????, 2: ?????????????????, 3: ?????????????????, 4: ?????????????????}
21
22 evaluation = evaluate_circuit(circuit, g_tables, inputs)

```

```

1  """
2  once you've found the input labels which make the circuit return `true`,
3  then concatenate them together, hash them,
4  and xor with the provided string to obtain the flag
5  """

```

- obtain\_flag.py: o objetivo, é necessário preencher o dicionário inputs com os rótulos corretos, o script então:

```

24 # circuit should return `true`
25 for i in circuit['outputs']:
26     assert evaluation[i] == keys[i][1]

```

- verifica se esses rótulos produzem a saída True

```

28 msg = "{}: {}: {}: {}".format(inputs[1], inputs[2], inputs[3], inputs[4])
29 msg = msg.encode('ascii')

```

- concatena os rótulos encontrados

```

31 m = hashlib.sha512()
32 m.update(msg)
33 m.digest()

```

- calcula o hash SHA512 dessa string

```

35 xor_flag = b'\x90',u'\x1b\x1dE:\xa8q\x91}&\xc7\x90\xbb\xce]\xf5\x17\x89\xd7\xfa\x07\x86\x83\xfa\x9b^\xcb\xd77\x00w\xca\xceXD7'
36
37 print( xor(m.digest(), xor_flag) )
38
39 assert xor(m.digest(), xor_flag) == flag

```

- faz um XOR do hash com um valor pré-definido (xor\_flag) para revelar a flag

```

1  from yao import GarbledCircuit
2  import json
3
4
5  circuit_filename = "circuit.json"
6  with open(circuit_filename) as json_file:
7      circuit = json.load(json_file)
8
9  # creates a new garbled circuit each time
10 gc = GarbledCircuit(circuit)
11
12 g_tables = gc.get_garbled_tables()
13 keys = gc.get_keys()
14
15
16 print("g_tables = {}".format(repr(g_tables)))
17 print("\nkeys = {}".format(repr(keys)))

```

```

11 import json
12
13 from yao import evaluate_circuit
14 from generate_garbled_circuit import g_tables, keys
15
16
17 circuit_filename = "circuit.json"
18 with open(circuit_filename) as json_file:
19     circuit = json.load(json_file)
20
21
22 inputs = {}
23 for i in circuit["inputs"]:
24     v = keys[i][1]
25     inputs[i] = v
26
27 evaluation = evaluate_circuit(circuit, g_tables, inputs)
28 print("")
29 print(evaluation)

```

- generate\_garbled\_circuit.py e evaluate\_garbled\_circuit\_example.py: são arquivos de exemplo que mostram como o circuito foi gerado e como seria avaliado se já

tivéssemos as chaves, não contém os segredos do desafio, servindo apenas para propósitos didáticos

## 5. Vulnerabilidade

No desafio proposto, é possível perceber que existem vulnerabilidades a serem exploradas para o ataque, as principais exploradas são:

**-Tamanho de chaves mal dimensionadas:** Chaves com apenas 24 bits, ou seja, as chaves são curtas e passíveis de força-bruta.

```
def generate_random_label():  
    return randrange(0, 2**24)
```

**Impacto:** reduz drasticamente a segurança; ataques práticos possíveis (força bruta / MITM).

**-Encriptação de um plaintext conhecido:**

O garbling envia um ciphertext que é a encriptação de um valor conhecido (0) para cada combinação de entradas.

No código *validation = encrypt(0, key0, key1)*, esse ciphertext serve como ponto fixo (C) para o ataque meet-in-the-middle. Sem esse ciphertext de referência, o atacante não teria um alvo simples para buscar colisões e o custo do ataque aumentaria muito.

```
gl = encrypt(key2, key0, key1)  
validation = encrypt(0, key0, key1)  
return (gl, validation)
```

**Impacto:** Reduz o custo do ataque de  $2^{48}$  e transforma uma segurança teoricamente forte em uma vulnerabilidade prática e explorável

**-Dupla encriptação determinística (sem uso de mecanismos de variação como nonce/tweak)**

A cifra aplica *encrypt\_data* duas vezes:

Como não há nonce/tweak, a encriptação é determinística (mesma mensagem + mesmas chaves → mesmo ciphertext), o que viabiliza um ataque *meet-in-the-middle*.

```
def encrypt(data, key1, key2):  
    encrypted = encrypt_data(data, key1)  
    encrypted = encrypt_data(encrypted, key2)  
    return encrypted
```

**Impacto:** Permite pré-cálculos e colisões exploradas pelo MITM.

### -Reutilização de primitive/chave entre gates (sem derivação por gate)

Reutilizar a mesma função/mesma forma de encriptar entre gates, sem identificação por gate, facilita tabelas e lookups pré-computados. Ou seja, usar a mesma função de encriptação para todos os gates permite pré-computar tabelas globais, facilitando ataques.

```
def encrypt(data, key1, key2):  
    encrypted = encrypt_data(data, key1)  
    encrypted = encrypt_data(encrypted, key2)  
    return encrypted
```

**Impacto:** Permite que um atacante crie uma tabela global e reutilize para qualquer gate, sendo que se cada gate tivesse subchaves derivadas ele precisaria recalculá-las por gate tornando muito caro.

## 6. Estratégia de exploração

### a. Passo a passo do ataque

- i. Analisar o circuito: identificar todas as portas do circuito e a topologia, determinando quais portas têm tabelas vulneráveis.
- ii. Separar a cifragem das portas vulneráveis: para cada porta alvo, separar a cifragem em camadas de cada label de entrada.
- iii. Construir a tabela hash do lado A: decifrar a primeira camada de cada candidato de label e armazenar resultados em uma tabela hash
- iv. Testar candidatos do lado B: decifrar a segunda camada e comparar com a tabela hash; pares que coincidem são consistentes.

- v. Propagar resultados para outras portas: usar os labels de saída como entradas para portas subsequentes e repetir o processo.
- vi. Gerar combinações consistentes: combinar candidatos de todas as portas para formar conjuntos válidos de labels para o circuito inteiro.

## **b. recuperando a flag**

- i. O script `obtain_flag.py` foi adaptado para ser executável, removendo suas dependências de servidor. Para isso, foram comentadas a importação do módulo `private_data` e as cláusulas `assert` correspondentes.
- ii. Os input labels recuperados na fase de ataque foram inseridos no dicionário `inputs` do script modificado, substituindo os valores placeholder.
- iii. A função `evaluate_circuit` foi executada com as tabelas (`g_tables`) e os inputs corretos. A avaliação resultou no label de saída correspondente ao valor `true`, validando a solução encontrada.
- iv. Os inputs foram formatados em uma única string, codificados para bytes e processados com o algoritmo de hash SHA512 para gerar um digest (resumo criptográfico).
- v. A flag foi decodificada através de uma operação XOR entre o digest do hash e a variável estática `xor_flag` fornecida no script, revertendo a ofuscação final.
- vi. A execução do script modificado resultou na impressão da flag final no terminal, concluindo o desafio.

## **7. Implementação**

### **Implementação passo a passo**

#### **3.1 Construção da tabela de encriptação de zero**

Criamos um dicionário `enc0_table` com todas as combinações:

`ciphertext = encrypt_data(0, key)` para `key` em  $[0, 2^{24})$ .

Esse mapeamento permite, durante o ataque, verificar se um mid-value obtido ao decriptar um `validation_ct` corresponde à encriptação de zero por alguma chave.

```
enc0_table = { encrypt_data(0, k): k for k in range(2**24) }
```

### 3.2 MITM

Após criar o dicionário `encryptions_of_zero` com todas as possibilidades de chave encriptada de zero, para cada parte A da tabela, testamos se um valor da `g_table` é semelhante, e armazenamos essa chave left como um possível candidato de todos os valores Right que são mapeados por mid, em um dicionário `Z_map`.

```
print('[!] generating lookup table...')
ENCRYPTIONS_OF_ZERO = defaultdict(list)
for key in tqdm(range(2**24)):
    ct = encrypt_data(0, key)
    ENCRYPTIONS_OF_ZERO[ct].append(key)
```

### 3.3 Recuperação da quadrupla

Com os quatro dicionários `Z_maps` construídos, foi feita uma busca cruzada:

Procuramos  $(a_0, a_1, b_0, b_1)$  tais que:

$$\text{decrypt}(c_1, a_0, b_0) == \text{decrypt}(c_2, a_1, b_0)$$
$$== \text{decrypt}(c_3, a_0, b_1)$$
$$== \text{decrypt}(c_4, a_1, b_1)$$

- Quando uma quadrupla consistente é encontrada, ela corresponde aos *input labels* corretos.
- Essa consistência dos inputs para reconstruir o mesmo valor de output, que elimina os falsos positivos mapeados no `Z_map`.

### 3.4 Cálculo da flag

Com  $(a_0, a_1, b_0, b_1)$  em mãos:

Montamos a string `f"{a0}:{a1}:{b0}:{b1}"`.

Aplicamos SHA-512.

Fizemos XOR byte a byte com `xor_flag`.

O resultado é a flag, validada contra o valor esperado no pacote.

```
pypy3 obtain_flag.py
[+] Entradas válidas (perm #1): {1: 11693387, 2: 11338704, 3: 7371799, 4: 2815776}

[*] msg: 11693387:11338704:7371799:2815776
[*] SHA-512(msg): f4404f106053754df728fe2854f4cfefa66c9b7cc0b9cc58d1eacdd301bb85077467a9fe69714a27a886b783f85140b417f9103abb3e664ef1584fec6a56b280
[*] FLAG (bytes): b'dice{N0w_YoUr3_Th1nkIn6_wi7H_pR0t0c015}'
[*] FLAG (ascii): dice{N0w_YoUr3_Th1nkIn6_wi7H_pR0t0c015}
```

## 8. Conclusão

A resolução deste desafio foi alcançada explorando falhas críticas na implementação do protocolo de Garbled Circuits, não em sua teoria. Vulnerabilidades como chaves de 24 bits e criptografia determinística permitiram um ataque de meet-in-the-middle, que nos levou a reverter o processo e recuperar os labels de entrada secretos. Este exercício prático demonstrou que a segurança teórica de um sistema é ineficaz sem uma implementação robusta, reforçando que a segurança real depende diretamente da qualidade do código e do design prático.

## Referências

Joseph, "DiceCTF 2021 - garbled" writeup: <https://jsur.in/posts/2021-02-08-dicectf-2021-garbled>

[https://en.wikipedia.org/wiki/Garbled\\_circuit](https://en.wikipedia.org/wiki/Garbled_circuit)

[https://en.wikipedia.org/wiki/Yao%27s\\_Millionaires%27\\_problem](https://en.wikipedia.org/wiki/Yao%27s_Millionaires%27_problem)

## SLIDES:

[https://www.canva.com/design/DAGzLai2dsY/R70qK9-LV0T9fJIjS66kwQ/edit?utm\\_content=DAGzLai2dsY&utm\\_campaign=designshare&utm\\_medium=link2&utm\\_source=sharebutton](https://www.canva.com/design/DAGzLai2dsY/R70qK9-LV0T9fJIjS66kwQ/edit?utm_content=DAGzLai2dsY&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton)

