# TP2_Ex1

November 10, 2025

## 0.1 ## Lógica Computacional: 25/26

## 0.2 TP2 - Ex1

*Grupo* 05

- Vasco Ferreira Leite (A108399)
- Gustavo da Silva Faria (A108575)
-

### 0.3 Afonso Henrique Cerqueira Leal (A108472)

## 0.4 Problema:

Construa uma resolução das seguintes questões a partir de "inputs" do problema: os parâmetros $\kappa$, $n$ e de probabilidade de falha $\varepsilon$ restrita apenas às "gates" and.

---

#### 0.4.1 Tarefa 1:

Construa algoritmos para, sob "inputs" do segredo $z \in \{0,1\}^n$ e da "chave mestra" $s \in \{0,1\}^\kappa$, construa o circuito. Adicionalmente a partir deste circuito , construa o modelo SMT do circuito com falhas.

#### 0.4.2 Tarefa 2:

Usando o modelo acima, tente construir uma possível estimativa para $z$ numa execução com falhas não nulas; isto é, encontrar

#### 0.4.3 Tarefa 3:

Conhecido $z \in \{0,1\}^n$ pretende-se maximizar a probabilidade de falhas and sem que o "output" $0^n$ seja alterado.

---

#### 0.4.4 Variáveis:

`n:` dimensão do problema, representa em bits a dimensão de z;

`k:` dimensão em bits de s;

`z:` vetor que representa o segredo;

`s:` vetor que representa a "chave mestra";

`lista:` representa os parâmentros do circuito (o, a, b,c) sendo o-offset e a,b,c vetores de coeficientes;

`x_bits:` representa o ('x') do circuito;

`falhas:` lista de booleanos que contém as falhas;

`saidas:` lista de tuplos que contém bits de valor e representação de falhas.

### 0.5 Definição Global

[ ]:
```
from z3 import *
import numpy as np

n = int(input("Escreva o n: ")) #n=200 #teste
k = int(input("Escreve o k: ")) #k=512 #teste

rng=np.random.default_rng(12345)
z=rng.integers(low=0, high=2, size=n, dtype=np.uint8)
s=rng.integers(low=0, high=2, size=k, dtype=np.uint8)
```

### 0.6 Função: produto_int

Calcula o produto interno de dois vetores

[ ]:
```
def produto_int(a, b):
    assert len(a)==len(b)
    n=len(a)
    res=0
    for i in range(n):
        prod=a[i]&b[i]
        res=res^prod
    return res
```

### 0.7 Funções: gate_xor, gate_and & gate_maj

Constrói as gates usadas para contruir o circuito.

`gate_xor:` modelo de uma gate XOR. É usada para combinar os termos da tua equação final;

`gate_and:` modelo de uma gate AND. Ponto onde se analisa falhas;

`gate_maj:` modelo de uma gate MAJ. Implementa a redundância tripla do circuito, escolhendo a 'maioria'.

[ ]:
```
def gate_xor(t1, t2, name):
    w1, d1=t1
    w2, d2=t2
```

```
    w=BitVec(name, 1)
    ww=w1^w2
    d=Or(d1, d2)

    rest=Or(d, w==ww)
    return (w, d), rest

def gate_and(t1, t2, falha, name):
    w1, d1=t1
    w2, d2=t2

    w=BitVec(name, 1)
    ww=w1&w2
    d=Or(d1, d2, falha) #falha aqui pq se ela falha entao a saida tbm

    rest=Or(d, w==ww)
    return (w, d), rest

def gate_maj(t1, t2, t3, name):
    w1, d1=t1
    w2, d2=t2
    w3, d3=t3

    w=BitVec(name, 1)
    maj=(w1&w2) | (w1&w3) | (w2&w3)
    d=Or(d1, d2, d3)

    rest=Or(d, w==maj)
    return (w, d), rest
```

### 0.8 Função: gate_prod

Modelo que faz o produto interno. Diferente de prod_int, esta opera com variáveis BitVec do z3

[ ]:
```
def gate_prod(vec_x, x_bits):
    assert len(vec_x)==len(x_bits)
    tam=len(vec_x)

    lista=[]

    for i in range(tam):
        bit=BitVecVal(int(vec_x[i]), 1)
        x_bit=x_bits[i]
        lista.append(bit & x_bit)

    if not lista:
        res=BitVecVal(0, 1)
```

```
    else:
        res=lista[0]
        for i in range(1, tam):
            res=res^lista[i]

    return (res, BoolVal(False))
```

### 0.9 Função: build_smt_model

Constrói o circuito dentro do z3 e do solver. Faz uso das gates definidas anteriormente para retornar variáveis essenciais.

[ ]:
```
def build_smt_model(solver, n, lista):
    x_bits=[]
    for i in range(n):
        x_bits.append(BitVec(f'x_{i}', 1))

    #reverter porque bits menos significativos primeiro
    x_input=Concat(list(reversed(x_bits)))

    falhas=[]
    saidas=[]

    for i in range(n):
        o, a, b, c=lista[i]

        o_wd=(BitVecVal(o, 1), BoolVal(False))

        a_x_wd=gate_prod(a, x_bits)
        b_x_wd=gate_prod(b, x_bits)
        c_x_wd=gate_prod(c, x_bits)

        #bandeiras de falha 'and'
        and1=Bool(f'and1_{i}')
        and2=Bool(f'and2_{i}')
        and3=Bool(f'and3_{i}')
        falhas.extend([and1, and2, and3])

        and1_wd, c1=gate_and(b_x_wd, c_x_wd, and1, f'and1_w_{i}')
        and2_wd, c2=gate_and(b_x_wd, c_x_wd, and2, f'and2_w_{i}')
        and3_wd, c3=gate_and(b_x_wd, c_x_wd, and3, f'and3_w_{i}')

        solver.add(c1)
        solver.add(c2)
        solver.add(c3)
```

```
        maj_wd, c_maj=gate_maj(and1_wd, and2_wd, and3_wd, f'maj_w_{i}')
        solver.add(c_maj)

        quadrado_maj=maj_wd

        #o ^ (a . x)
        xor_wd1, c_xor1=gate_xor(o_wd, a_x_wd, f'xor_A_{i}')
        xor_wd2, c_xor2=gate_xor(xor_wd1, quadrado_maj, f'xor_B_{i}')

        solver.add(c_xor1)
        solver.add(c_xor2)

        saidas.append(xor_wd2)

    print(f"\nModelo SMT (re)construído com {type(solver)}.")
    print(f"  - {n} variáveis 'x_bits' criadas.")
    print(f"  - {len(falhas)} variáveis 'falhas' criadas.")
    print(f"  - {len(saidas)} variáveis 'saidas' criadas.")

    #retorna as variaveis que precisamos de usar fora
    return x_bits, falhas, saidas, x_input
```

[ ]:
```
def main():
    semente_s=np.random.SeedSequence(s.tolist())
    rng_s=np.random.default_rng(semente_s)
    sub_seeds=rng_s.integers(low=0, high=2**64, size=n, dtype=np.uint64)
    lista =[]
    for i in range(n):
        rng_sub=np.random.default_rng(sub_seeds[i])
        a=rng_sub.integers(0, 2, size=n, dtype=np.uint8)
        b=rng_sub.integers(0, 2, size=n, dtype=np.uint8)
        c=rng_sub.integers(0, 2, size=n, dtype=np.uint8)

        a_z=produto_int(a, z)
        b_z=produto_int(b, z)
        c_z=produto_int(c, z)
        o=a_z ^ (b_z & c_z)

        lista.append((int(o), a, b, c))

    print("-------------------------------------------------")
    print(f"Geração de parâmetros concluída.")
    print(f"Total de conjuntos de parâmetros gerados: {len(lista)}")


    print("\n--- Ponto 2: Encontrar um 'falso segredo' z' ---")
```

```
    solver_p2 = Solver()
    x_bits_p2, falhas_p2, saidas_p2, x_input_p2=build_smt_model(solver_p2, n,␣
↪lista)

    print("A adicionar restrição: Saída (w) == 0.")
    for (w, d) in saidas_p2:
        solver_p2.add(w==BitVecVal(0,1))

    print("A adicionar restrição: Pelo menos uma falha.")
    solver_p2.add(Or(falhas_p2))

    z_int = 0
    for i in range(n):
        z_int += int(z[i]) * (2**i)
    z_original_z3 = BitVecVal(z_int, n)

    print("A adicionar restrição: Input z' != z.")
    solver_p2.add(x_input_p2 != z_original_z3)

    print("A verificar o solver (solver_p2.check())...")
    check_p2 = solver_p2.check()

    if check_p2 == sat:
        print("\n -> SATISFAZIVEL: Encontrada uma solução!")
        m_p2 = solver_p2.model()

        #temos de extrair o 'z' (que são os x_bits) ANTES de imprimir o z␣
↪original
        z_prime_list = [m_p2.eval(x_bits_p2[i]).as_long() for i in range(n)]
        z_prime = np.array(z_prime_list, dtype=np.uint8)

        print(f"  - Segredo Original (z) : {z}")
        print(f"  - Estimativa (z')      : {z_prime}")

        if np.array_equal(z, z_prime):
            print("  (Nota: A estimativa z' é IGUAL ao segredo original z.)")
        else:
            print("  (Nota: Encontrado z' DIFERENTE do segredo original!)")

        falhas_ocorridas_p2 = []
        for f in falhas_p2:
            if m_p2.eval(f):
                falhas_ocorridas_p2.append(str(f))

        print(f"\n  - Total de falhas 'and' ocorridas:␣
↪{len(falhas_ocorridas_p2)}")
```

```
        # print(f"  - Lista de falhas: {falhas_ocorridas_p2}") #descomentar pa␣
↪ver

    elif check_p2 == unsat:
        print("\n -> INSATISFAZIVEL.")

    else:
        print(f"\n O Solver retornou: {check_p2}")


    print("\n--- Ponto 3: Maximizar falhas com 'z' conhecido ---")

    opt = Optimize()

    x_bits_p3, falhas_p3, saidas_p3, _ = build_smt_model(opt, n, lista)

    print("A adicionar restrição: Input 'x' deve ser o segredo 'z'.")
    for i in range(n):
        opt.add(x_bits_p3[i] == int(z[i]))

    print("A adicionar restrição: Saída (w) deve ser 0.")
    for (w, d) in saidas_p3:
        opt.add(w==BitVecVal(0,1))

    num_falhas = Sum([If(f, 1, 0) for f in falhas_p3])

    print("A definir objetivo: Maximizar o número total de falhas 'and'.")
    opt.maximize(num_falhas)

    print("A verificar o otimizador (opt.check())...")
    check_p3 = opt.check()

    if check_p3 == sat:
        print("\n -> SATISFAZIVEL: Encontrada uma solução ótima!")
        m_p3 = opt.model()
        max_falhas = m_p3.eval(num_falhas).as_long()

        print(f"  - Input 'x'           : {z} (o segredo original, como␣
↪forçado)")
        print(f"  - Saída do circuito   : 0^n (como forçado)")
        print(f"  - NÚMERO MÁXIMO DE FALHAS: {max_falhas}")

        falhas_ocorridas_p3 = []
        for f in falhas_p3:
            if m_p3.eval(f):
                falhas_ocorridas_p3.append(str(f))
```

```
        print(f"  - Total de {len(falhas_ocorridas_p3)} falhas ativadas (de␣
 ↪{len(falhas_p3)} possíveis).")

    elif check_p3 == unsat:
        print("\n -> INSATISFAZIVEL: Não há solução.")

    else:
        print(f"\n O Otimizador retornou: {check_p3}")


if __name__ == "__main__":
    main()
```