Trabalho Prático 1 - Segurança Computacional

Guilherme Nonato da Silva - 221002020

December 20, 2024

1 Introdução

No presente trabalho, analisaremos um código em Python, capaz de encriptar e decriptar usando o algoritmo S-DES. O código foi feito dividindo as diferentes etapas e operações do algoritmo em funções, para que o processo seja mais claro e as diferentes operações mais simples.

A versão de Python que foi usada é a 3.11.9, e o código fonte será disponibilizado junto a este documento em um único arquivo *zip*, e também em link para seu repositório do GitHub ao final do documento.

2 Algoritmo S-DES

S-DES, ou *Simplified Data Encryption Standard*, é um algoritmo de criptografia baseado no DES (*Data Encryption Standard*) e simplificado para fins acadêmicos. Ele faz uso de manipulação de bits, como permutação e operações lógicas *bit a bit*, além de rodadas de Feistel para criptografar um bloco de código de 8 bits usando uma chave de 10 bits.

A forma que as permutações serão feitas são a partir de uma única função, a `perm()`, que usa listas pré definidas para definir a ordem das permutações. Além disso, as `SBoxes` também são definidas previamente, com os números em decimal para uma melhor visualização.

```
P4 =  [1,3,2,0]
P8 =  [5,2,6,3,7,4,9,8]
P10 = [2,4,1,6,3,9,0,8,7,5]
IP =  [1,5,2,0,3,7,4,6]
IPI = [3,0,2,4,6,1,7,5]
EP =  [3,0,1,2,1,2,3,0]

S0 = [[1,0,3,2],
      [3,2,1,0],
      [0,2,1,3],
      [3,1,3,2]]

S1 = [[0,1,2,3],
      [2,0,1,3],
      [3,0,1,0],
      [2,1,0,3]]
```

Como especificado no documento de proposta do trabalho, usaremos a seguinte chave e bloco de bits para realizar o S-DES e verificar o resultado ao final. As definimos como strings para facilitar a construção das funções.

```
# Resultado esperado da criptografia: 10101000

chave = "1010000010"
bloco = "11010111"
```

2.1 Geração de Chave

A primeira etapa conta com a função principal `genChave()`, que recebe a chave de entrada como argumento.

```
def genChave(chave):
  chave_perm = perm(chave, P10)
  left, right = chave_perm[:5], chave_perm[5:]

  k1 = ''.join(perm(shift(left, 1) + shift(right, 1), P8))
  k2 = ''.join(perm(shift(left, 3) + shift(right, 3), P8))
  return k1, k2
```

Ela realiza a geração de chave utilizando duas outras funções para fazer as devidas permutações e *shifts*. Primeiramente, permuta a chave de entrada usando `P10` e então a divide em dois blocos de 5 bits, para então fazer os *Left Shifts* e aplicar a permutação com `P8` usando ambas as metades. Ao invés de fazer um *Left Shift* de 1 bit para gerar `k1` e então um de 2 bits neste para gerar `k2`, fazemos um de 3 bits diretamente para evitar dividir o processo mais ainda.

Estas são as funções auxiliares de `genChave()`:

```
def perm(bits, lista):
  return [bits[i] for i in lista]

def shift(bits, n):
  return bits[n:] + bits[:n]
```

A função `perm()` usa das listas de permutação definidas anteriormente para saber quais bits mudar de lugar, e a `shift()` reparte e reconecta os bits a partir do tamanho do *Left Shift* para atingir os *shifts* desejados.

Seguindo o algoritmo até aqui, as subchaves `k1` e `k2` devem ser 10100100 e 01000011:

```
# Output de print("Subchaves:", k1, k2), na funcao encSDES()
# Subchaves: 10100100 01000011
```

2.2 Criptografia

A função principal dessa fase é a `encSDES()`, que usa mais 3 funções auxiliares (fora as da geração de chave) para realizar a criptografia do bloco de entrada de 8 bits.

```
def encSDES(bits, chave):
  k1, k2 = genChave(chave)

  bits_perm = perm(bits, IP)
  parte1 = fk(bits_perm, k1)
  parte2 = fk(switch(parte1), k2)
  return perm(parte2, IPI)
```

Assim como é descrito no algoritmo, após a geração das subchaves, o bloco de bits de entrada é permutado a partir da lista `IP`, e então é usada na função `fk()`, junto da primeira subchave (`k1`), onde são realizadas as rodadas de Feistel.

```
def fk(bits, chave):
  left, right = ''.join(bits[:4]), ''.join(bits[4:])
  right_exp = ''.join(perm(right, EP))

  xor = '{:08b}'.format(int(right_exp, 2) ^ int(chave, 2))
  sbox = sBox(xor[:4], S0) + sBox(xor[4:], S1)

  resultado = ''.join(perm(sbox, P4))
  return '{:04b}'.format(int(left, 2) ^ int(resultado, 2)) + right
```

A função `fk()` é a mais complexa entre as outras, apesar de ser relativamente pequena. Ela recebe o bloco de bits já permutado e a subchave correspondente, dividindo-a em dois para fazer a operação de `xor` e a função `F` (Não é uma função definida neste código, mas sim na descrição do algoritmo S-DES). A função F compreende a expansão da metade da chave sendo operada usando a lista de permutação EP (linha 3), o `xor` *bit a bit* da chave com esta metade expandida (linha 5; o valor é convertido para um valor binário de 8 bits), e então a substituição com SBox e a permutação final dos 4 bits resultantes (linhas 6 e 8, respectivamente).

Esta última parte utiliza uma outra das funções auxiliares, a `sBox()`, que recebe os 4 bits que definirão as coordenadas do número escolhido e a respectiva SBox a ser utilizada.

```
def sBox(bits, sbox):
  linha = int(bits[0]+bits[3], 2)
  coluna = int(bits[1]+bits[2], 2)
  return '{:02b}'.format(sbox[linha][coluna])
```

Aqui, os valores em binário dos respectivos bits são convertidos para `int`, para poderem determinar o índice dentro da matriz, e é na saída da função que o número dentro da matriz é localizado e convertido para binário.

Após isso, o mesmo caminho (agora em `parte2` usando k2) é feito pelo bloco de bits manipulado pela função `switch()`, que apenas troca de lugar as duas partições, para que a parte direita dos bits (`right`) também seja manipulada pela função.

Por fim, a função `encSDES()` retorna o resultado permutado com a lista `IPI`, que é a inversa de IP, para que o fluxo de criptografia seja concluído. Neste ponto, com a chave e bloco de entrada fornecidos, o resultado da criptografia deve ser 10101000. Usamos as variáveis `chave` e `bloco` que declaramos anteriormente para iniciar a criptografia, e vamos alimentar as próximas funções com elas.

```
# Resultado esperado da criptografia: 10101000

bloco = encSDES(bloco, chave)

print("Bloco criptografado:", ''.join(bloco))

# Output
# Bloco criptografado: 10101000
```

2.3 Descriptografia

Com todo o resto realizado, a descriptografia se torna trivial. A função `decSDES()` é praticamente idêntica a `encSDES()`, salvo a ordem das chaves, que deve ser invertida (seguindo o diagrama do algoritmo) para corretamente retornar o resultado correto, que é o bloco de entrada de 8 bits (11010111).

```
def decSDES(bits, chave):
  k1, k2 = genChave(chave)

  bits_perm = perm(bits, IP)
  parte1 = fk(bits_perm, k2)
  parte2 = fk(switch(parte1), k1)
  return perm(parte2, IPI)

# Resultado esperado da descriptografia: 11010111
bloco = decSDES(bloco, chave)
print("Bloco descriptografado:", ''.join(bloco))

# Output
# Bloco descriptografado: 11010111
```

3 Conclusão

Vimos que o código é capaz de corretamente criptografar e descriptografar o bloco de bits fornecido seguindo os parâmetros do algoritmo S-DES. O código conta com funções que seguem o fluxo de criptografia descrito no algoritmo. O código fonte será disponibilizado junto a este documento em um arquivo *zip*, mas também segue o repositório no GitHub como outro meio de acesso.

References

Appendix G - Simplified DES. Stallings, William (2010)