Criptografia moderna

Trabalho 2, Grupo 3

Gustavo Zambonin, Ranieri Schroeder Althoff

Introdução à Criptoanálise (UFSC-INE5451)

Questão 1

a. A tabela de aproximações lineares apresenta todas as tendências para cada combinação de variáveis aleatórias, com respeito à respectiva caixa-S do cifrador. Estas tendências são calculadas da seguinte forma: $(y_1, y_2, y_3, y_4) = \pi(x_1, x_2, x_3, x_4)$ e $\oplus_{i=1}^{4} a_i x_i \oplus b_i y_i = 0$, ou seja, se a operação $\oplus$ entre as variáveis de entrada e saída for igual à 0, um contador é incrementado. As melhores tendências são as que distam de 8, ou seja, da probabilidade usual.

```
1  def bin_to_str(num):
2      return '{0:04b}'.format(num)
3
4
5  def multi_xor(arg1, arg2):
6      result = 0
7      for i in range(len(arg2)):
8          if int(arg2[i]) == 1:
9              result = result ^ int(arg1[i])
10     return result
11
12
13 nl = []
14 for input_sum in range(16):
15     row = []
16     for output_sum in range(16):
17         count = 0
18         for key in list(sbox.keys()):
19             if multi_xor(bin_to_str(input_sum), key) ^ \
20                multi_xor(bin_to_str(output_sum), sbox[key]) == 0:
21                 count += 1
22         row.append(count)
23     nl.append(row)
```

- Nota: a caixa-S foi omitida do código. O código completo encontra-se aqui.
- Linhas 7–9: responsável por realizar a operação $\oplus$ sobre múltiplos bits, especializada para esta implementação, visto que o processo só é fundamentado se um bit válido ocorrer.
- Linhas 19–20: realiza a operação $\oplus$ entre os resultados do $\oplus$ entre as variáveis de entrada e saída com relação à cada permutação da caixa-S.

A tabela resultante segue abaixo.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	16	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
1	8	10	6	8	10	8	8	6	4	6	6	8	10	8	4	10
2	8	10	8	10	6	8	6	8	6	8	10	4	4	6	8	10
3	8	8	10	10	8	12	10	6	6	6	8	8	10	6	12	8
4	8	10	8	6	8	10	8	6	10	4	10	8	6	8	6	4
5	8	12	6	6	10	10	8	12	6	10	8	8	8	8	10	6
6	8	8	12	8	10	10	6	10	8	8	8	12	6	6	6	10
7	8	6	6	8	12	6	10	8	8	6	6	8	4	6	10	8
8	8	10	10	8	8	6	6	8	10	8	4	6	10	4	8	6
9	8	8	8	12	10	10	6	10	10	6	6	6	8	12	8	8
A	8	12	10	10	6	6	12	8	8	8	6	10	6	10	8	8
B	8	6	12	6	8	6	8	10	4	6	8	6	8	10	8	6
C	8	8	10	10	12	8	10	6	8	12	10	6	8	8	6	6
D	8	6	8	6	6	12	10	8	8	10	4	6	6	8	6	8
E	8	6	6	12	6	8	8	10	6	8	8	10	8	6	6	4
F	8	8	8	8	8	8	12	12	10	6	10	6	10	6	6	10

{gustavo.zambonin,ranieri.althoff}@grad.ufsc.br

src/

b. A aproximação linear consiste em seguir a variável escolhida no percurso da SPN, e sabendo que a variável aleatória a ser analisada é $\mathbf{X}_{16} \oplus \mathbf{U}_1^4 \oplus \mathbf{U}_9^4$, a entrada em $\mathbf{S}_4^1$ é igual a $0001_2 = 1_{10}$. Isto denota a variável a em $N_L(a, b)$. b, por outro lado, pode ser escolhido de modo que auxilie na linearidade das permutações de bits. Nesta caixa-S, na linha 1, $b = 8$ dispersa menos bits do que $b = E$, portanto a tendência nesta parte do processo é de $\epsilon(1, 8) = \frac{N_L(1,8)}{16} - \frac{1}{2} = -\frac{1}{4}$. Este bit será mapeado para $\mathbf{S}_1^2 = 0001_2 = 1_{10}$, e novamente $b = 8$ será escolhido, e sua tendência nesse ponto será $-\frac{1}{4}$ novamente. Em $\mathbf{S}_1^3$, na linha 8 da tabela (pois sua entrada é 1000_2), os candidatos de melhor tendência são $b = A$ e $b = D$, onde o primeiro ativa um número menor de caixas-S, e portanto será escolhido (ademais, estes bits, na saída da caixa-S, estão presentes na variável aleatória supracitada): $\epsilon(8, 10) = -\frac{1}{4}$. Aplicando o lema do empilhamento de Matsui: $2^{k-1} \prod_{j=1}^{k} \epsilon_{i_j} = 2^{3-1} \prod_{j=1}^{3} -\frac{1}{4} = -\frac{1}{16}$.

c. A justificativa do ataque linear à SPN é semelhante ao ataque diferencial, no sentido de ambos usarem bits do texto em claro e bits antes da última caixa-S. Para tal, precisa-se de variáveis aleatórias que contenham tais bits — o que acontece neste caso. Esta aproximação incluirá três caixas-S ativas:

$$\begin{aligned}\mathbf{T}_1 \oplus \mathbf{T}_2 \oplus \mathbf{T}_3 &= (\mathbf{U}_{16}^1 \oplus \mathbf{V}_{13}^1) \oplus (\mathbf{U}_4^2 \oplus \mathbf{V}_1^2) \oplus (\mathbf{U}_1^3 \oplus \mathbf{V}_1^3 \oplus \mathbf{U}_4^3) \\ &= (\mathbf{X}_{16} \oplus \mathbf{K}_{16}^1 \oplus \mathbf{V}_{13}^1) \oplus (\mathbf{V}_{13}^1 \oplus \mathbf{K}_4^2 \oplus \mathbf{V}_1^2) \oplus (\mathbf{V}_1^2 \oplus \mathbf{K}_1^3 \oplus \mathbf{V}_1^3 \oplus \mathbf{V}_3^3)\end{aligned}$$

- Excluindo os bits de chave, pois estes são fixos e não influenciarão no módulo da tendência, a variável aleatória proposta e sua tendência são confirmadas:
- $\mathbf{X}_{16} \oplus \mathbf{U}_1^4 \oplus \mathbf{U}_9^4 = \pm\frac{1}{16}$.
- Os bits que poderão ser obtidos ao final do ataque são $\mathbf{K}_{(1)}^5$ e $\mathbf{K}_{(3)}^5$. Existem 2^8 possibilidades para este conjunto de bits, com um contador atrelado a cada candidato.
- Por fim, um contador será mantido para cada ocorrência de $\mathbf{X}_{16} \oplus \mathbf{U}_1^4 \oplus \mathbf{U}_9^4 = 0$. Como esta variável aleatória dista de $\frac{T}{2}$ por $\pm\frac{T}{16}$, o contador para esta variável se destacará no conjunto, e portanto retornará os bits corretos em parte das vezes, dependente do número de pares de texto cifrado e decifrado.

d. A implementação do raciocínio acima, que funciona com τ até 1500 textos, é discutida abaixo.

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from random import randint, shuffle
from spn import SPN

key = '{0:032b}'.format(randint(0, 2**32))
permut = {
    1: 1, 2: 5, 3: 9, 4: 13, 5: 2, 6: 6, 7: 10, 8: 14, 9: 3,
    10: 7, 11: 11, 12: 15, 13: 4, 14: 8, 15: 12, 16: 16,
}
sbox = {
    '0000': '1000', '0001': '0100', '0010': '0010', '0011': '0001',
    '0100': '1100', '0101': '0110', '0110': '0011', '0111': '1101',
    '1000': '1010', '1001': '0101', '1010': '1110', '1011': '0111',
    '1100': '1111', '1101': '1011', '1110': '1001', '1111': '0000',
}
inverse_sbox = {v: k for k, v in sbox.items()}
_spn = SPN(permut, sbox)


def gen_pairs(k, range_):
    seq = list(range(2**16))
    shuffle(seq)
    pairs = []

    for i in seq[:range_]:
        i = '{0:016b}'.format(i)
        pairs += [[i, _spn.encode(i, key)]]
    return pairs


count = [[0 for i in range(16)] for j in range(16)]

for pair in gen_pairs(key, 1500):
    for i in range(16):
        for j in range(16):
            v4_1 = i ^ int(pair[1][0:4], 2)
            v4_3 = j ^ int(pair[1][8:12], 2)
            u4_1 = inverse_sbox['{0:04b}'.format(v4_1)]
            u4_3 = inverse_sbox['{0:04b}'.format(v4_3)]
            z = int(pair[0][-1]) ^ int(u4_1[0]) ^ int(u4_3[0])
            if z == 0:
                count[i][j] += 1
```

```
45
46  subkey = _spn.gen_subkeys(key)[4]
47  print("subkey:\t{} {}".format(subkey[:4], subkey[8:12]))
48  _max = max(map(max, count))
49  for n, l in enumerate(count):
50      try:
51          print("guess:\t{0:04b} {0:04b}".format(n, l.index(_max)))
52      except:
53          pass
```

- Linha 7: gera uma chave de 32 bits aleatória.
- Linhas 23–29: gera 2^{16} textos e escolhe 1500 destes aleatoriamente, cifrando-os com a chave escolhida.
- Linhas 38–44: lógica de contadores para cada candidato às subchaves.
- Linha 48: filtra o contador que mais se destacou e mostra as subchaves correspondentes.

Questão 3

O RSA é um cifrador baseado em criptografia com chave pública. A dificuldade de quebrá-lo está centrada no uso de números primos grandes para codificar mensagens e a inabilidade computacional (até o presente momento) de fatorar tais números. Uma chave é composta por cinco elementos: n e seus fatores primos p e q, e $a * b \equiv 1 \pmod{\phi(n)}$. Escolhe-se, cuidadosamente, dois primos p e q (tem-se então $n = p * q$), b tal que $mdc(b, \phi(n)) = 1$, e calcula-se $a = b^{-1} \pmod{\phi(n)}$. n e b são públicos, enquanto p, q e a são secretos. As funções de codificação e decodificação, respectivamente, são as seguintes: $x^b \pmod{n}$ e $y^a \pmod{n}$ para x, $y \in Z_n$. Abaixo, segue a implementação de um decodificador RSA, dados os números públicos n e b.

```
1   def decode26(enc_num):
2       coefficients = []
3       while enc_num:
4           coefficients.append(enc_num % 26)
5           enc_num //= 26
6
7       still_enc = ""
8       for num in coefficients[::-1]:
9           still_enc += chr(num + 65)
10
11      return still_enc
12
13
14  def phi(n):
15      p = factor(n)
16      return (p - 1) * ((n / p) - 1)
17
18
19  def rsa_dec(text=enc_text, n=31313, b=4913):
20      final_text = ""
21      a = inv_mod(b, phi(n))
22      for i in text:
23          final_text += decode26(quad_mult(i, a, n))
24      return final_text
```

- Nota: o vetor que contém o texto codificado foi omitido. O código completo encontra-se aqui.
- Linha 1: o método de decodificação é pertinente a esta implementação de RSA, e por isto não foi movido para o arquivo de utilidades.
- Linhas 4–5: cria um vetor que contém os coeficientes da função que codifica numericamente as palavras. Ele será invertido por conta do modo que as operações matemáticas são conduzidas.
- Linhas 8–9: itera sobre a inversão do vetor acima, para que as letras estejam na ordem correta, e calcula a posição do número na codificação ASCII, assim retornando uma palavra legível.
- Linhas 15–16: implementação da função totiente de Euler ($\phi(n)$), que usa o método $p-1$, discutido abaixo, para descobrir os fatores de um número primo, e assim retornar $(p-1) * (q-1)$.
- Linha 21: computa $a = b^{-1} \pmod{\phi(n)}$.
- Linha 23: executa o processo $i^a \pmod{n}$, onde i é cada um dos pedaços de texto codificados, com o algoritmo quadrado-e-multiplica. Depois, aplica a função de decodificação supracitada para gerar as letras correspondentes.

O resultado final foi modificado apenas para adição de espaços. O texto original foi adicionado abaixo para comparação de conteúdo, visto que a pontuação e ênfase contextual podem ser ambíguas.

```
LAKE WOBEGON IS MOSTLY POOR SNDY SOIL AND EVERY SPRING THE EARTH HEAVES UP A
NEW CROP OF ROCKS PILES OF ROCKS TEN FEET HIGH IN THE CORNERS OF FIELDS PICKED
BY GENERATIONS OF US MONUMENTS TO OUR INDUSTRY OUR NCESTORS CHOSE THE PLACE
TIRED FROM THEIR LONG JOURNEYS D FOR HVING LEFT THE MOTHERLAND BEHIND AND THIS
PLACE REMINDED THEM OF THERE SO THEY SETTLED HERE FORGETTING THAT THEY HAD
LEFT THERE BECAUSE THE LND WASNT SO GOOD SO THE NEW LIFE TURNED OUT TO BE LOT
LIKE THE OLD EXCEPT THE WINTERS RE WORSEZ
```

Lake Wobegon is mostly poor sandy soil, and every spring the earth heaves up a new crop of rocks. Piles of rocks ten feet high in the corners of fields, picked by generations of us, monuments to our industry. Our ancestors chose the place, tired from their long journey, sad for having left the motherland behind, and this place reminded them of there, so they settled here, forgetting that they had left there because the land wasn't so good. So the new life turned out to be a lot like the old, except the winters are worse. [1]

Questão 5

O método de fatoração de inteiros de Pollard, também conhecido como método $p-1$, é utilizado para fatorar números até um certo número de dígitos, pois sua complexidade torna-se impraticável quando fatores muito grandes tentam ser calculados. Tem como base, na aritmética modular, o Teorema de Fermat (mais especificamente a expressão $a^{p-1} \equiv 1 \pmod{p}$). A ideia do algoritmo é encontrar um fator $p-1 \mid B!$ para algum B pequeno. Utilizando Fermat, conclui-se que $2^{B!} \equiv 1 \pmod{p}$, o que significa que $p \mid (2^{B!} - 1)$. Como $p \mid n$ também se aplica, pois p é um fator de n, então por definição $p \mid mdc(2^{B!} - 1, n)$ existe, e esse segundo número será o resultado desejado. Segue abaixo uma pequena implementação do algoritmo.

```
1  def factor(n=618240007109027021):
2      a = 2
3      bound = int(n**.5)
4
5      for j in range(2, bound):
6          a = a**j % n
7          d = gcd(a - 1, n)
8          if 1 < d < n:
9              return d
```

- Linha 2: inicia-se o processo com 2, pois é o primeiro fator não-trivial que pode influenciar no resultado.
- Linha 5: calcula iterações de a, até o limite B, que podem resultar num fator válido para o número original.
- Linha 7: o processo de MDC pode ser realizado uma vez a cada cálculo de a ou uma vez ao final do algoritmo, não influenciando no resultado final.
- Linha 8: checa se d está no intervalo válido. Se não existir um fator, o programa não retornará nada.

Ao final do processamento, tem-se como resposta o fator **250387201**. Dividindo o número original por este fator, tem-se uma divisão inteira com quociente **2469135821**.

Questão 6

Outro algoritmo para fatoração de inteiros foi concebido por Dixon. Baseia-se em encontrar certas congruências do estilo $x^2 \equiv y^2 \pmod{n}$, pois se $n \mid (x-y)(x+y)$, então $mdc(x-y, n)$ será um bom fator de n. Utilizando uma base de primos "pequenos", uma iteração é realizada de modo a achar tais congruências e, por fim, realizar o processo de máximo divisor comum sobre estas. A implementação é discutida abaixo.

```
1  def factor(n=256961):
2      base = [-1] + [x for x in range(2, 32) if is_prime(x)]
3      start = int(n**.5)
4      pairs = []
5
6      for i in range(start, n):
7          for j in base:
8              if i**2 % n == j**2 % n:
9                  pairs.append([i, j])
10
11     for i in pairs:
12         factor = gcd(i[0] - i[1], n)
13         if factor != 1:
14             return factor, n // factor
```

- Linha 2: define a base de fatores primos com um teste de primalidade simples.

- Linhas 6–7: definem o número de iterações a ser feitas (todos os números entre $\sqrt{n}$ e n comparados com todos os números da base de fatores).
- Linhas 8–9: testa a congruência característica e armazena o par de inteiros para o processo de MDC posterior.
- Linhas 12–14: todos os valores são submetidos ao MDC característico, e caso um deles retorne um fator não-trivial, o processo termina.

Assim, o algoritmo retornou o fator **293**, resultado do processamento das congruências, e **897**, resultado da divisão do número original por seu fator descoberto. É possível que, dependendo da congruência a ser escolhida (se esta for válida), o primeiro fator a ser encontrado será diferente.

Questão 7

A fatoração de grandes números inteiros não é o único processo matemático no qual cifradores são baseados. O logaritmo discreto descreve uma estrutura difícil de ser quebrada se seu elemento primitivo for cuidadosamente escolhido (dado um $\beta \in Z_p^*$, encontrar um expoente único $0 \leq a \leq p-2$ tal que $\alpha^a \equiv \beta \pmod{p}$). Um algoritmo para a resolução deste logaritmo, comumente chamado de *baby-step giant-step*, separa o expoente a em α^{mj} e $\beta\alpha^{-i}$, utilizando a equação $\log_\alpha \beta = i * m + j$, com $m = \lceil \sqrt{p-1} \rceil$ e $0 \leq i, j < m$ como fundamento. Dividindo o problema, torna-se mais fácil compor o expoente a. A implementação da estratégia será dirimida abaixo.

```
1  def shanks_algorithm(alpha=6, beta=248388, p=458009):
2      m = int(p**.5) + 1
3      y1 = alpha**m % p
4      y2 = inv_mod(alpha, p)
5      frst_list, scnd_list = [], []
6
7      for i in range(m):
8          frst_list.append([i, y1**i % p])
9          scnd_list.append([i, beta * (y2**i) % p])
10
11     for i in frst_list:
12         for j in scnd_list:
13             if i[1] == j[1]:
14                 return m * i[0] + j[0]
```

- Linhas 3–4: computa valores que serão usados repetidamente ($\alpha^m \pmod{p}$ e a inversa modular de α).
- Linhas 8–9: o algoritmo busca encontrar j e i tal que seus respectivos pares sejam $\alpha^{mj} \pmod{p}$ e $\beta\alpha^{-i} \pmod{p}$, já que estes são iguais.
- Linhas 11–13: busca os elementos adjacentes às segundas coordenadas se estas forem iguais.
- Linha 14: resolve $\log_\alpha \beta = i * m + j$.

Por fim, o resultado de $\log_6 248388$ em $Z^*_{458009} = a = \mathbf{232836}$. Isto pode ser verificado aplicando o número encontrado no logaritmo discreto. De fato, $6^{232836} \equiv 248388 \pmod{458009}$, pois $6^{232836} \pmod{458009} = 248388$.

Referências

[1] Garrison Keillor. *Lake Wobegon Days*. Penguin Books, 1985.