Criptografia RSA
Pré-processamento

Junior R. Ribeiro

27 de outubro de 2021

1 Tabelas de Caracteres

O computador trabalha apenas com números. Tudo o que você vê e ouve a partir de um dispositivo eletrônico vem de uma fonte numérica. Inclusive textos.

Quando você vê um texto escrito na tela, o programa faz uma conversão dos números em caracteres. Existem algumas tabelas de conversão, dentre as quais se destacam a ASCII e a UNICODE.

1.1 Tabela ASCII

A tabela ASCII é bem simples, comportando um total de 128 diferentes caracteres não acentuados. Aqui, os caracteres podem ser categorizados da seguinte maneira:

imprimíveis: as letras, dígitos e pontuações diversas. Como exemplo, o caracter @ é imprimível.

não imprimíveis: espaçamento, tabulações, quebras de linha e outros. São escritos usando um padrão de 2 caracteres, que geralmente é a contrabarra "\" e uma das seguintes letras $\{a, b, t, n, v, f, r\}$, como por exemplo \n indicando uma quebra de linha, décimo caractere da tabela.

controle: caracteres especiais utilizados principalmente pelo sistema operacional para trabalhar com armazenamento e leitura de dados de diferentes fontes, bem como para trafegar dados via internet. Os caracteres de controle geralmente não são diretamente suportados pelas linguagens de programação, pois podem causar algum efeito colateral no sistema operacional e portanto precisam ser acessados de forma indireta, usando alguma biblioteca específica.

As principais linguagens de programação dão suporte direto aos caracteres imprimíveis e não imprimíveis. Caracteres de controle também não são imprimíveis.

1.2 Tabela UNICODE

Como o número de caracteres da ASCII é muito limitado, não é possível ter suporte a outros idiomas que não o Inglês usando essa tabela. Foi criada uma tabela pensando no suporte a todos os idiomas, contando com quase 138.000 caracteres, desde ideogramas chineses, emojis, caracteres árabes, matemáticos, de música e vários outros. Esta tabela é uma extensão da ASCII, portanto todos os caracteres ASCII fazem parte da UNICODE e ocupam as mesmas posições, ou seja, as primeiras 128 posições.

Alguns exemplos: as peças de xadrez se situam no intervalo $[9812, 9823]$; os caracteres Braile no intervalo $[10240, 10495]$; os emojis podem ser vistos aqui, mas seus códigos estão em base hexadecimal e alguns deles são grupos de mais de um código. Tabela para o árabe. Tabela para o coreano.

2 Pré-processamento

Como visto no Episódio 05, as mensagens que serão codificadas usando a Criptografia RSA não devem ultrapassar o tamanho da chave pública, e também percebemos que não pode ser um número muito pequeno. Por esse motivo, precisamos quebrar a mensagem em blocos menores que a chave pública e que sejam números grandes.

Vamos desenvolver a seguir uma estratégia para codificar mensagens simples, que utilizam a tabela ASCII. Uma ideia semelhante pode ser desenvolvida para incluir mais caracteres, tais como os da tabela UNICODE.

★ **Na verdade, o procedimento que faremos será válido para mais caracteres do que somente os ASCII.**

2.1 Estratégia

Vamos seguir os seguintes passos. Vamos prosseguir o exemplo usando a mensagem

msg = "amo matemática"

1. A primeira etapa do pré-processamento é utilizar as tabelas ASCII ou UNICODE, conforme o caso, e mapear cada caracter da mensagem para seu respectivo código numérico. Aplicando esta etapa à nossa mensagem, obtemos

   msg = [97, 109, 111, 32, 109, 97, 116, 101, 109, 225, 116, 105, 99, 97]

2. A tabela ASCII comporta caracteres numerados de 0 a 127, ou seja, com 1, 2 ou 3 dígitos. Isso é um problema pois, para os cálculos do computador, 000 é o mesmo que 00 e o mesmo que 0. Esse problema surge porque **zero à esquerda** não conta, como no caso do primeiro código, 97, que é o mesmo que 097. Precisamos unificar essa situação. Uma forma de fazê-lo é **somar 100 aos códigos dos caracteres.** Assim, o intervalo mudará de $[0, 127]$ para $[100, 227]$, todos com 3 dígitos e sem ambiguidades.

Perceba que mais códigos são permitidos com essa ideia, todo o intervalo de 128 até 899 pode ser aproveitado, pois esses números somados a 100 nos dão 228 e 999, e o próximo número, 1000, tem 4 dígitos, levando-nos novamente a ambiguidade. Esse é o motivo do aviso ★ acima.

Após somar 100, todos os códigos passam a ter 3 dígitos, conforme mencionado:

msg = [197, 209, 211, 132, 209, 197, 216, 201, 209, 325, 216, 205, 199, 197]

3. Após somar 100 ao código de cada um dos caracteres que originalmente pertencem ao intervalo 0 a 899, vamos concatená-los (juntá-los). Mas antes disso, precisamos determinar quantos códigos devemos concatenar de modo que o resultado seja menor que a chave pública e de modo que a mensagem seja suficientemente grante.

   (a) Suponha que nossa chave pública seja um número com 16 dígitos. Percebemos que todos os códigos que temos têm todos 3 dígitos cada um. Podemos determinar quantos caracteres colocar em cada bloco fazendo uma continha simples e arredondar para baixo. Para ter certeza de que o bloco seja menor que a chave pública, podemos ainda subtrair uma unidade do resultado.

$$N_{(chars-per-block)} = \lfloor 16 \div 3 \rfloor - 1 = \lfloor 5.3333 \rfloor - 1 = 5 - 1 = 4.$$

   Assim, cada bloco conterá 4 caracteres da mensagem, e podemos dividir a mensagem em blocos.

   msg = [197, 209, 211, 132 @ 209, 197, 216, 201 @ 209, 325, 216, 205 @ 199, 197]

   (b) Veja que o último bloco contém apenas 2 caracteres, o que pode resultar em uma mensagem muito pequena. Podemos adicionar caracteres falsos para fazer a mensagem ficar maior (mas devemos ter o cuidado de removê-los na decodificação da mensagem!). Uma ideia possível é adicionar zeros à esquerda, e adicionar 1 à esquerda desses zeros. No caso, ficaria algo assim.

   msg = [197, 209, 211, 132 @ 209, 197, 216, 201 @ 209, 325, 216, 205 @ 100, 000 199, 197]

   Na decodificação, vamos olhar de 3 em 3 dígitos e, se houver uma sequência com mais de 3 zeros no começo do bloco antecedido por 1, podemos descartar todos eles.

   Agora falta apenas concatenar os códigos para obter os blocos.

   msg = [197209211132, 209197216201, 209325216205, 100000199197]

   Esta é a nossa mensagem pré-processada, pensando-se em uma chave pública de 16 dígitos. Agora cada bloco precisa ser codificado individualmente e ser enviado separadamente. SIM! Não podemos misturar os blocos, nem mudar sua ordem, senão não teremos como obter de volta a mensagem original.

A implementação de (3b) precisa ser cuidadosa, por isso, não faremos na nossa implementação.

IMPLEMENTATION IN PYTHON PROGRAMMING LANGUAGE

PUBLIC KEY (module)

```
pubkey =
0x4c95b17188c5a64ed4073ff22120f733d1c1fc422ad5e76e983426d6f6676631336ea36a1
dbbf98c0aad2abde2fd0097dcd4bb6179eb778e26c36d43fe07dc09ddd3b9cb164fba6acd22
2726d83689aa5bb3b6e5f6c80a4b2b7ba15d372831772430a1cd836300544e2d03d2b58f076
f85910dc8ebe4ffe7125a85a6e7a10f62f9fc3d9e3d1485b32339f5f4a9ef97053213883e11
6d21eb97a073e5b47e847f92ff384f7f617992f039a1018d9069ec8692d65dd828390a6d3f3
949899bc02e187b47e9f325843d2954be3d6626d7b91ff9ca86d567ffe10962a60526ae3f3b
e2956b00cd39eb151e7a969af33c4969bcb224969f0a21033d8ae8405d47ad404fcd56ab39f
1dc41ab6f2b3b1931bd78e1170ca26fcc30f6a5665580ff1478126e10dd0a06cb78b8aed884
157da13883c8b1897dd43af7055989980ca515f33c542456b7041b4b2b274e9fc28817d643f
384321f4e72c253eaa3bab0dfdb629883464e5bf9ac5cc2871fea00eb7de1cd708a390a462e
49dbe25f1269c03db0d05db3836d74516c914b9fb3182761541d3d57a3d15347bcb00ffcbda
094cf3c728e228fa2a7e437eddb01d69495e9d4290b85c1b23c2300c6ee1e4b2bfc4f926ae7
25be85a65045336cee3ce79da161387809aa95775b5fb13eca0e502576761dd1db43eadc7a2
aa625aa10e75e7968c075c054f7f6b41bace718243e79dee30c605b358fd7953988814bdb14
5b06f29fbfd8b86c283650ed1c83f8477394509ea19ac9a19485d817a78d435c4ef5171f2cc
a529d9064626ec21edb03a8b4851a0e3cbc0def54aca2c3315f137573576c52fe81d5a40173
d583579003b697d43e5161c680a5dc239bf9c6102098b883eff97f76a536c9f01f132329ade
f6136f7dc5173b831e84045380d834242524c9937c69188ecf4b0b028a604b7e430a542d52e
dc61484fce638986788938b70e92ddeb5c738bbb78809e8365987f694677383db588d10061c
52250eac28f3377fb9f34d01a5f541f03e33a0708be7d8e60de13e47c52b6609b7ee4b7f55e
8c66a3d80035b705724875f0ea19efbaf615ff455dd88de669c71f2a13beb04eb443de47f38
a001caaa20bfcd4b582043f9646e6b5700308780ab8733b4e0d3a91010a736eb950789ee3af
76b9ed04d07684000364234d8876a7b357a889c77c675f62ff471d5ecaaefe2d3c0decbaec6
bff4f13df4a21a993397445ec92fa9166ec737690f565bbeb6ab35697879aad0892fb91
```

ENCODING EXPONENT

```
exponent = 0x49b
```

PRIVATE KEY (exponent for decoding)

```
privkey =
0x214217a49a10c8c3c78ec3d4a12b31cfb7ce7cbf265c75015807039f246fc7a0fba0bd15f
b1b90a190aff34b4bc63e5b920ea60e1a74961022344c1ca88238097ef2a2c296d35ea2544e
3bfea1349007f8ec1deb92e77b3df4eb36396aae3986778177fd2eda0093c87d826b925f8d1
bcd6fd8d295c6ad3adfccfbb92c59eb52b7b03313dcdbf72f2d7aa9c7663dec08560492e326
9e2167653d36058472defebabbb2d5e00796b8f87f835e4240e2a76bbae18b1d6196ecdc47c
0499fdb69d2718d79c63884acf116822b18b596d6d7734984767b821003eca0177588df51f5
f643ed53f1c47853dd77762a895496deaa19af0c364cebcfe69d2a04110ed7815460fd08910
```

```
55e214b88a66c9bfc69bb555f57c7bc2aa423059a17ab51068023cbeaa961811ae983a98396
f1aa49ebd0230adf6d24cba7da5ee749a12cfbda523814fdec07dcf3b2553fe41a496e73322
2f72ebc9acb21cc493c073f90c405784c1da9bcad908d08435d8c0d6d6e7834bcb4c065bcc4
69e653f98469da52cf7f9db9df2e7259a49dc4579761fe6d93ead333bd0b370e6f095d25046
5dee9bce20f4208f6145d0722761fa194fd92ecbea9973c72bf9221c6a581c18a4ebcb84f66
1d861889e03608d9338f859f7496df9f219303d4c8ea69db98b463f5f38dc56fac8f002bf2e
34d35b65d707708ba266fdc2a53a333fc7af22d996082bee06f21e3b5168d96f72ac43492c2
1e9af5b1a66e955ae8b002667da4048d59d265ad6feff5a9b9dd967e047c8d80b72624f69de
9137c449f6f81fd9f50b26d20eef280e97956c01290c48d1283a158a62d48d08f7f2f23e4b4
644da86c68e64d2c62cdbfffd1197c2836b109634e3442111e5d1d07bda46d7a1271e5b16d8
1857f915ed610f97ba46db654d0eed909f468a86ff2c8183a486e72d056337e3bc8ddf0bc86
b361eb89e3ff02dcb05ccab167989758846eb53a57a2e3a1ae222c05a1b29ee810fbf1ddc12
09755630b178fb788157e3a284892abb3aa542118e6918137c1ef4ba6aeb6dda506c7652d58
cce4ab415f7ca93f614bc2ce76fd2e2c911b5edcd0fdc54f543522360cc506253c3cc3197ef
cef3f34eb707ceff891d4857b69779f4ac2210df55570d897e5fdb66664da5ea7678e8c69f9
1f1a8338333e087cba231f7560b23053aa77c66c52e085b49871761fe17eaaf60d75c98f5dd
82961991be42cb1de013b6c541a74dffaf15cb8d4cf4302b983f8591edb51f25d9c33
```

ENCODING AND DECODING

```
block = some_number_less_than_pubkey
encoded_block = pow(block, exponent, pubkey) # encoding
decoded_block = pow(encoded_block, privkey, pubkey) # decoding
```

PRE-PROCESSING, ENCODING AND DECODING

```
# file rsa.py
__all__ = ["encode", "decode"]

def msg_between_0_and_899_to_blocks(msg, pubkey_length):
    # adding 100
    ls = [ord(char)+100 for char in msg]  # int values
    blocks = []
    # number of chars per block
    charsPblock = (pubkey_length//3) - 1

    while True:
        if len(ls) > charsPblock:
            new_block = int("".join([str(i) for i in ls[0:charsPblock]]))
            ls = ls[charsPblock:]
            blocks.append(new_block)
        else:
            new_block = int("".join([str(i) for i in ls]))
            blocks.append(new_block)
            break
    return blocks
```

```
def blocks_to_msg_between_0_and_899(blocks):
    msg = ""
    for block in blocks:
        string = str(block)
        ls = []
        while string:
            # subtracting 100
            ls.append(chr(int(string[0:3])-100))
            string = string[3:]
        msg += "".join(ls)
    return msg


def encode(msg, exponent, pubkey):
    pubkey_length = len(str(pubkey))
    blocks = msg_between_0_and_899_to_blocks(msg, pubkey_length)
    return [pow(blk, exponent, pubkey) for blk in blocks]


def decode(blocks, privkey, pubkey):
    decoded_blocks = [pow(blk, privkey, pubkey) for blk in blocks]
    return blocks_to_msg_between_0_and_899(decoded_blocks)
```