

Rivest Shamir Adleman

História

A história do algoritmo RSA começou no *Massachusetts Institute of Technology* (MIT) nos anos 70 quando um matemático, um criptólogo e um cientista da computação se juntaram - Ronald Rivest, Adi Shamir e Leonard Adleman respectivamente. RSA é derivado das iniciais dos sobrenomes de cada um deles.

O algoritmo RSA é uma das formas mais comuns de criptografia assimétrica, que utiliza um par de chaves: uma pública e uma privada. A chave pública pode ser compartilhada livremente, enquanto a chave privada é mantida em segredo pelo proprietário.

A ideia básica por trás do RSA é que é fácil calcular o produto de dois números primos grandes, mas é extremamente difícil, para qualquer pessoa que não possua a fatorização, descobrir os fatores primos originais desses produtos.

Criptografia

Criptografia pode ser vista como a prática ou a ciência de proteger informação.

A sua história tem as raízes na antiguidade, com civilizações como os egípcios e os babilônios que desenvolveram técnicas rudimentares. Durante a época medieval, a criptografia desempenhou um papel crucial nas comunicações militares e diplomáticas. O Renascimento viu avanços significativos, incluindo a cifra de *Vigenère*. Durante o século XIX foram surgindo cifras mais complexas como a cifra de *Playfair* ou de *Hill*. Nas guerras mundiais e na Guerra Fria, houve também grandes progressos no desenvolvimento de novas cifras e de máquinas criptográficas, como a *Enigma*.

Com o mundo digital, a criptografia tornou-se essencial para a segurança da informação e das transações *online*, com algoritmos modernos como o RSA e o AES.

Criptografia Simétrica

Apesar de ser discutível, um dos exemplos mais simples de criptografia são as cifras nomeadas de “*one-time pad*” ou cifras de bloco único em português. São cifras que utilizam uma chave para encriptar e desencriptar uma mensagem em que a chave só pode ser utilizada uma vez, caso contrário será possível fazer análises estatísticas aos textos cifrados para descobrir o conteúdo original.

Vamos imaginar duas personagens, Alice e Bob (são vistos várias vezes em textos sobre criptografia), que querem comunicar em segredo um com o outro. A Alice quer enviar uma mensagem para o Bob. Para simplificar a explicação vamos considerar que a mensagem foi transformada em binário (0's e 1's). A Alice e o Bob encontram-se pessoalmente e definem uma chave secreta (também em binário). Digamos que a chave combinada pelos dois é a seguinte: “1101001”.

A Alice quer encriptar a seguinte mensagem: “1011010”. Para aplicar a encriptação foi combinado entre os dois que seria o algoritmo do XOR (*Exclusive Or*) da lógica booleana. O XOR resumidamente recebe dois *bits* e analisa a sua igualdade. Se os *bits* A e B forem iguais então o resultado Q será 0. Se os *bits* de entrada forem diferentes então Q será 1.

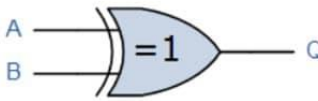
Symbol	Truth Table		
 2-input Ex-OR Gate	A	B	Q
	0	0	0
	0	1	1
	1	0	1
	1	1	0
Boolean Expression $Q = A \text{ XOR } B$			

Figura 1 - Exclusive Or

Apliquemos então o XOR entre a mensagem da Alice e a chave definida pelos dois para encriptar a mensagem.

Mensagem	1	0	1	1	0	1	0
Chave	1	1	0	1	0	0	1
Mensagem Cifrada (XOR)	0	1	1	0	0	1	1

Figura 2 - Encriptação da mensagem da Alice

A Alice agora pode enviar a mensagem encriptada para o Bob que pode desencriptar utilizando o mesmo método entre a mensagem encriptada e a chave (funciona, pode tentar desse lado).

Este método funciona, mas apresenta alguns problemas. Um dos maiores problemas é que a chave tem de ser do mesmo tamanho da mensagem. Outro é que a Alice e o Bob tiveram de se encontrar previamente para combinarem uma chave secreta (para isso mais valia passarem logo a mensagem normalmente) e outro problema está na segurança em que se a mesma chave for utilizada para várias mensagens então vão ser encontradas semelhanças nas mensagens cifradas.

Mensagem diferente	1	1	1	1	1	1	1
Chave	1	1	0	1	0	0	1
Mensagem Cifrada (XOR)	0	0	1	0	1	1	0

Figura 3 - Mensagens diferentes com chaves iguais podem originar resultados semelhantes

Com esta informação é possível fazer análises estatísticas para descobrir padrões entre as mensagens cifradas. Nas cifras de substituição de letras também é possível fazer a mesma análise já que em português por exemplo, umas letras e combinações de letras são mais populares do que outras. Ao utilizar a mesma chave para mensagens diferentes estamos a preservar alguns destes padrões e com a ajuda de um computador pode ser possível descobrir alguma informação senão mesmo toda a informação da mensagem original.

Criptografia Assimétrica

Quando usamos cifras do tipo “*one-time pad*”, usamos um sistema que é intrinsecamente “simétrico”, isto é, um sistema em que a chave serve tanto para encriptar como para desencriptar a informação, e cada indivíduo, Alice e Bob, poderia fazer ambas as operações (neste caso a operação acabaria por ser a mesma por causa de como o XOR funciona).

E se houvesse um sistema em que isso não acontecesse, em que o processo de encriptação é distinto do de desencriptação? Ganharíamos algo com isso?

Imaginemos agora outra situação diferente em que a Alice quer enviar ao Bob algo contrabandeado. A Alice e o Bob vivem num sistema político totalitário onde todo o correio é interceptado pelo governo. No entanto, a Alice e o Bob têm cada um deles um aloquete único em que apenas eles têm as respetivas chaves. Os aloquetes são inquebráveis e se o governo não os conseguir abrir então a encomenda é enviada para o destinatário. Como é que a Alice consegue enviar o contrabando para o Bob?

A Alice pode colocar o conteúdo dentro de uma caixa e fechar a caixa com o seu aloquete. A caixa fechada é enviada e se for interceptada o governo não a consegue abrir porque não tem a chave da Alice. Quando a caixa chega ao Bob ele também não a consegue abrir porque também não tem a chave da Alice, mas ele pode colocar o seu aloquete junto ao dela. O Bob envia novamente a caixa fechada e o governo continua sem a conseguir abrir. A Alice recebe, retira o seu aloquete (mas também não consegue abrir pois não tem a chave do aloquete do Bob) e envia de volta mais uma vez. Agora, quando o Bob a receber já pode abrir o seu aloquete e ter acesso ao conteúdo da caixa.

No mundo da criptografia são utilizadas funções matemáticas em vez de aloquetes físicos. Para este propósito, podem existir múltiplas cópias do “aloquete” em que o Bob pode dar a quem ele quiser que lhe envie uma mensagem. Simplificando ainda mais o processo, o Bob envia uma cópia de como funciona o seu “aloquete” para a Alice e ela aplica o algoritmo na mensagem e envia-a de volta para ele (relembrando que a chave do Bob continua secreta e uma vez que a Alice encripta a mensagem com o algoritmo dele, ela não conseguirá desencriptá-la nem ninguém para além do Bob).

Resumo

A Alice quer enviar uma mensagem ao Bob. Ela diz-lhe que precisa de lhe passar uma mensagem secreta e o Bob vai criar um par de chaves, uma pública e uma privada e vai-lhe enviar a chave pública. A Alice com a chave pública vai encriptar a mensagem e enviar para o Bob. Ele através da chave privada que apenas ele sabe qual é, consegue desencriptar a mensagem.

Chave pública de Bob

Quando se está a escolher uma operação para aplicar chaves, procura-se algo que seja relativamente fácil de calcular numa direção e difícil na outra. Deve certamente haver vários métodos de o fazer, mas o RSA utiliza a factorização de números primos.

Relembrando que um número é considerado primo, se e só se, for divisível por 1 e por si próprio. Dados dois números primos, é fácil fazer a multiplicação entre eles, mas é muito difícil partir do seu produto e obter a sua factorização. É como se estivéssemos a fazer uma receita de um bolo e tentar retirar cada um dos ingredientes de uma mistura homogénea.

Até aos dias de hoje, ainda não foi descoberta uma forma de conseguir obter os números iniciais que seja melhor do que um *brute force*. Para termos uma ideia do tempo necessário para factorizar os números, com por exemplo 300 dígitos, levaria cerca de 100 mil anos. De facto, este método mostra-se computacionalmente inquebrável com números de tais dimensões já que a força do algoritmo está directamente relacionada com o número de *bits* da chave.

Voltando à chave do Bob. O Bob vai escolher dois números primos aleatoriamente, p e q . Números primos minimamente bons teriam de ter no mínimo cerca de 250 *bits* que dá cerca de 60 caracteres, mas para simplificar a explicação e demonstração vamos utilizar números mais pequenos.

Sejam os números primos de Bob $p = 11$ e $q = 13$, ele vai multiplicá-los e obter o primeiro número da chave pública. Seja $n = pq$, tal que $n = 143$ (n é o chamado “módulos”). Para melhorar o aspeto da explicação vamos colocar os valores em tabelas.

Variáveis	Valores
p	11
q	13
n	143

Figura 4 – Variáveis de Bob

Agora que temos n definido precisamos da segunda parte da chave pública que se traduz na variável e (e é o chamado “expoente”).

Função totiente de Euler

A função totiente de Euler denotada por $\phi(n)$ (lê-se “phi de n ”), na teoria dos números representa a quantidade de números naturais que são relativamente primos com n não excedendo n .

$$\text{Phi de } n \rightarrow \phi(n) = |\{k \in \mathbb{N}: 1 \leq k \leq n \text{ e } \text{mdc}(k, n) = 1\}|$$

Por exemplo, para calcular $\phi(8)$ basta pegar no conjunto de números de 1 a 8 e analisar quais são os números que são primos relativamente ao 8, ou seja, os números cujo máximo divisor comum entre ele e 8 seja 1.

Analisando então o seguinte conjunto $\{\underline{1}, 2, \underline{3}, 4, \underline{5}, 6, \underline{7}, 8\}$ é possível verificar que apenas quatro números (sublinhados) são primos relativamente ao 8. Desta forma, $\phi(8) = 4$.

Para números normais a função $\phi(n)$ pode ser muito complexa, mas os números primos têm uma particularidade. Admitindo que p é um número primo, então $\phi(p) = p - 1$ pois todos os números naturais menores que p são primos relativamente a p .

Outra condição que se verifica é que, admitindo que p e q são números primos, então $\phi(pq) = (p - 1)(q - 1)$.

De volta à nossa chave pública, a variável e deve ser um número relativamente primo a $\phi(n)$. Ora, temos então que escolher um número que verifique a seguinte condição $\text{mdc}(e, (p - 1)(q - 1)) = 1$.

Em geral, o número e é escolhido também como um número primo e na maior parte dos casos $e = 65537$, mas para o nosso exemplo vamos utilizar números mais pequenos como o 13.

Se fizermos as contas verificamos que $\text{mdc}(13, (11 - 1)(13 - 1)) = 1$, logo 13 é um possível candidato para a variável e .

Variáveis	Valores
p	11
q	13
n	143
e	13

Figura 5 – Variáveis de Bob

Agora que temos a variável e definida, o Bob já pode enviar a chave pública para a Alice que é então o par de variáveis (e, n) , ou seja, $(13, 143)$.

Encriptação da Alice

A Alice quer enviar a mensagem “RSA” para o Bob. Através da chave pública que o Bob lhe forneceu $(13, 143)$, ela é agora capaz de encriptar a mensagem seguindo os passos abaixo:

1. **Codificar a mensagem de texto para o seu código ASCII e atribuir os valores a uma variável**

Em ASCII, $R = 82$, $S = 83$ e $A = 65$. Seja m a mensagem da Alice, $m = 828365$.

Variáveis	Valores
e	13
n	143
m	828365

Figura 6 - Variáveis da Alice

2. Aplicar a fórmula de encriptação

Para encriptar a mensagem, a Alice terá de aplicar a seguinte fórmula com as suas variáveis. Seja c a mensagem encriptada, $m^e \equiv c \pmod n$.

Por outras palavras, a Alice tem de calcular o resto da divisão entre m^e e n para descobrir o c .

NOTA: Temos de ter o cuidado de verificar se $m < n$. Se $m > n$ então vamos ter de dividir m em blocos mais pequenos de forma que sejam menores que n .

Como neste caso $m > n$ ($828365 > 143$), m vai ter de ser dividida em três blocos: 82, 83 e 65. Para cada bloco vai ser aplicada a mesma fórmula de encriptação.

- $R \quad 82^{13} \equiv 4 \pmod{143} \rightarrow c = 4$
- $S \quad 83^{13} \equiv 18 \pmod{143} \rightarrow c = 18$
- $A \quad 65^{13} \equiv 65 \pmod{143} \rightarrow c = 65$

Assim sendo, temos que $c = 41865$.

Variáveis	Valores
e	13
n	143
m	828365
c	41865

Figura 7 - Variáveis da Alice

A Alice conseguiu encriptar a sua mensagem através da chave pública, mas agora como faz o Bob para a desencriptar?

Chave privada de Bob

Para além das suas variáveis iniciais, Bob precisa de mais uma para formar a chave privada, a variável d .

Esta variável d traduz-se na chave que vai permitir calcular o inverso modular de e para conseguirmos reverter a função anteriormente aplicada.

Inverso Modular

Na matemática básica quando temos um número multiplicado pelo seu inverso o resultado é sempre 1. Por exemplo, seja a um número qualquer, o inverso de a é $\frac{1}{a}$ porque $a * \frac{1}{a} = 1$.

Já na matemática modular não temos uma operação de divisão para calcular o inverso. No entanto, temos o inverso modular que se traduz em encontrar um valor b tal que $a * b \equiv 1 \pmod{c}$.

Um método simples de fazer o cálculo é ir substituindo b por valores do intervalo $[1, c]$ já que fazer a multiplicação inicial com valores de $b > c$ é considerado redundante.

Para encontrar a nossa variável d da chave privada é só aplicar a mesma lógica com seguinte fórmula: $d * e \equiv 1 \pmod{\phi(n)}$.

Substituindo d pelos valores de 1 a 120, encontramos que o número 37 satisfaz a condição.

$37 * 13 \equiv 1 \pmod{120}$, por outras palavras, $481 \pmod{120} = 1$.

Variáveis	Valores
p	11
q	13
n	143
e	13
d	37
c	41865

Figura 8 - Variáveis de Bob

Descriptação de Bob

Agora que Bob tem todas as variáveis da chave privada (e, d), já consegue descriptar a mensagem que a Alice lhe enviou.

Aplicando novamente a aritmética modular, a fórmula de descriptação é a seguinte: $c^d \equiv m \pmod{n}$.

Como inicialmente a mensagem m teve de ser encriptada em três blocos diferentes por causa de $m > n$, a Alice teve, de alguma forma, de passar essa informação ao Bob. Num caso real, esta situação deverá ser evitada, no entanto, para o nosso exemplo vamos considerar que Bob já obteve essa informação e então conseguiu descriptar a mensagem da seguinte forma:

$$4^{37} \equiv m \pmod{143} \rightarrow m = 4^{37} \pmod{143} (=) m = 82 \rightarrow ASCII = R$$

$$18^{37} \equiv m \pmod{143} \rightarrow m = 18^{37} \pmod{143} (=) m = 83 \rightarrow ASCII = S$$

$$65^{37} \equiv m \pmod{143} \rightarrow m = 65^{37} \pmod{143} (=) m = 65 \rightarrow ASCII = A$$

Porquê que este algoritmo é seguro?

Enquanto fazia as pesquisas sobre o algoritmo e coloquei em prática o exemplo utilizado, reparei que é possível chegar à chave privada através da chave pública muito facilmente, no entanto, a minha teoria foi refutada rapidamente.

Sabendo que (d, n) é a chave privada, a única coisa que nos falta descobrir para calcular o d é $\phi(n)$, pelo que $d * e \equiv 1 \pmod{\phi(n)}$. Ora, sabendo que $\phi(n)$ corresponde a todos os números entre 1 e n , que são primos relativamente a n , com um simples *script* em *python* conseguiria calcular rapidamente o $\phi(n)$.

Primeira tentativa de *crack* ao RSA

A minha primeira ideia foi percorrer todos os números de 0 a n e calcular o *mdc* entre cada número e n . Desta forma, se o resultado do *mdc* entre o número i (cada número do intervalo utilizado) e n fosse 1, então teria um contador que aumentava de um em um sempre que essa condição se verificava, no fim retornava o contador.

```
1 import time
2 from math import gcd
3
4 def calcular_phin(n):
5     contador = 0
6     for i in range(1, n):
7
8         if gcd(i, n) == 1:
9             contador += 1
10
11     return contador
12
13 n = int(input("n > "))
14
15 start = time.time()
16 phin = calcular_phin(n)
17 end = time.time()
18
19 tempo_execucao = end - start
20
21 print("phi de n: {} \nTempo de execução: {}".format(phin, tempo_execucao))
```

Figura 9 – Código da primeira tentativa de *crack* ao RSA

Como seria de esperar, este código não demorou nem um segundo a ser executado. Admitindo que o meu computador não é nem de longe nem de perto um supercomputador (i5-12400F, 32GB de RAM), mas para um cálculo tão pequeno foi capaz de obter o resultado que pretendia.

```
n > 143
phi de n: 120
Tempo de execução: 0.0
```

Figura 10 – Primeiro teste da primeira tentativa de *crack* ao RSA

Como é possível verificar, o nosso $\phi(n)$ foi calculado rapidamente, mas isso foi porque o nosso n tinha apenas 8 *bits*. Vejamos agora outra situação.

Num outro teste em que utilizei um exemplo do site *PicoCTF* (exercício: “Mind your Ps and Qs”), em que o $n = 769457290801263793712740792519696786147248001$

937382943813345728685422050738403253 (269 bits), o resultado já não foi tão fácil de obter ...

```
n > 769457290801263793712740792519696786147248001937382943813345728685422050738403253
```

Figura 11 - Segundo teste da primeira tentativa de crack ao RSA

Após cerca de 12 horas de espera pelo resultado, parei a execução do programa pois já sabia que não iria a lado nenhum e a eletricidade não é propriamente barata para manter um computador a processar durante tanto tempo.

No entanto, se os números primos que foram originalmente utilizados para calcular n forem do nosso conhecimento, conseguimos utilizar a fórmula $\phi(n) = (p - 1)(q - 1)$.

Segunda tentativa de *crack* ao RSA

A segunda estratégia foi dividir o n em fatores primos e depois encontrar dois cuja multiplicação seja igual a n , ou seja, descobrir p e q .

```
1 from sympy import factorint
2 import time
3 from datetime import datetime
4 from colorama import Fore, Back, Style
5
6 numero = int(input("n > "))
7 print("Tempo início: ", datetime.now())
8
9 start = time.time()
10
11 fatores_primos = factorint(numero)
12 print(Fore.YELLOW + "* Dividido em fatores primos." + Style.RESET_ALL, datetime.now())
13
14 fatores_unicos = sorted(fatores_primos.keys(), reverse=True)
15 print(Fore.YELLOW + "* Fatores primos organizados." + Style.RESET_ALL, datetime.now())
16
17 fator1 = None
18 fator2 = None
19
20 print(Fore.YELLOW + "* A procurar fatores que satisfaçam o objetivo." + Style.RESET_ALL, datetime.now())
21 for fator in fatores_unicos:
22     if numero % fator == 0:
23         fator1 = fator
24         fator2 = numero // fator
25         break
26
27 end = time.time()
28 print(Style.RESET_ALL + "Tempo fim: ", datetime.now())
29 execution_time = end - start
30
31 if fator1 and fator2:
32     multiplicacao = fator1 * fator2
33
34     print(Fore.GREEN + "p =", fator1)
35     print(Fore.GREEN + "q =", fator2)
36     print(Fore.GREEN + "p * q =", multiplicacao)
37     print(Fore.GREEN + "p * q == n -->", multiplicacao == numero)
38     print(Style.RESET_ALL + "Tempo: ", execution_time)
39 else:
40     print(Fore.RED + "Não foi possível encontrar dois números primos cujo produto seja igual a" + Style.RESET_ALL, numero)
```

Figura 12 - Código da segunda tentativa de crack ao RSA

Novamente, com o nosso exemplo do Bob e da Alice foi possível obter o resultado pretendido rapidamente.

```

Tempo inicio: 2024-02-27 20:23:28.469079
* Dividido em fatores primos. 2024-02-27 20:23:28.469079
* Fatores primos organizados. 2024-02-27 20:23:28.469079
* A procurar fatores que satisfaçam o objetivo. 2024-02-27 20:23:28.469079
Tempo fim: 2024-02-27 20:23:28.469079
p = 13
q = 11
p * q = 143
p * q == n --> True
Tempo: 0.0

```

Figura 13 - Primeiro teste da segunda tentativa de crack ao RSA

No entanto, com o segundo teste aconteceu algo curioso. O n do segundo exemplo, como referido anteriormente, tem cerca de 269 bits. Após executar o programa com n com os 269 bits, reparei que demoraria horas a descobrir p e q , no entanto quando n tinha apenas cerca de 265 bits, ou seja, removi o último algarismo, obtive o seguinte resultado:

```

n > 76945729080126379371274079251969678614724800193738294381334572868542205073840325
Tempo inicio: 2024-02-27 20:24:21.838454
* Dividido em fatores primos. 2024-02-27 20:24:21.840451
* Fatores primos organizados. 2024-02-27 20:24:21.840451
* A procurar fatores que satisfaçam o objetivo. 2024-02-27 20:24:21.840451
Tempo fim: 2024-02-27 20:24:21.840451
p = 279802651200459561350087560916253376780817455249957434113943901340153472995783
q = 275
p * q = 76945729080126379371274079251969678614724800193738294381334572868542205073840325
p * q == n --> True
Tempo: 0.001997232437133789

```

Figura 14 – Segundo teste da segunda tentativa de crack ao RSA

Foi possível encontrar p e q daquele n em menos de um milissegundo. Assumindo que o programa está bem escrito, é incrível como apenas 4 bits fazem toda a diferença na segurança da informação.

Bem, mas se analisarmos matematicamente deixa de ser assim tão incrível pois se analisarmos todas as combinações possíveis com 269 e 265 bits ($2^{269} - 2^{265}$) vemos que existe uma diferença de 889283245342588380853025164866723132313113482232119531823034245180772835634708480 combinações possíveis, ou seja, está explicado o porquê de 4 bits fazerem tanta diferença.

Se 269 bits conseguem complicar tanto a vida a um computador, então imagine o que 512, 1024, 2048 e por aí fora, conseguiriam fazer. Não imagine, não tem tempo suficiente para isso.

Por esta razão é que o RSA é utilizado até aos dias de hoje e continua a ser seguro, porque fazer um ataque de *brute force* do género do que tentei fazer é impossível em tempo útil. Até mesmo os computadores mais avançados que existem atualmente precisariam de milhões de anos para processar tantos cálculos.

Factos interessantes sobre o RSA

Porque e é normalmente 65537?

O valor $e = 65537$ é muito comum de ser utilizado, e isto deve-se ao facto de que para além de ser um número primo, tem uma representação binário muito simples. 65537 em decimal é igual a 10000000000000001 em binário. Isto faz com que os cálculos sejam mais eficientes, o que aumenta muito a performance da computação das operações de criptografia.

Para além disso, $e = 65537$ satisfaz a necessidade de ser relativamente primo a $\phi(n)$ e de ser suficientemente grande para fornecer boas condições de segurança à cifra.

Onde é utilizado no dia a dia?

Este algoritmo é utilizado por nós todos os dias e nem sequer nos apercebemos disso. Por exemplo, quando acedemos a um *site* seguro, utilizamos o protocolo *HTTPS* que é o mesmo que dizer *HTTP over SSL*.

Ora, o HTTP (*Hypertext Transfer Protocol*) é um protocolo que permite estabelecer a ligação entre um *browser* e um servidor. É o que torna possível fazer um pedido de uma página *web* e que nos seja mostrada no nosso motor de busca. Com a evolução da internet, apareceu a necessidade de encriptar estas ligações de forma que ninguém conseguisse analisar uma rede e ver o que cada individuo pesquisava. A solução foi implementar o SSL (*Secure Socket Layer*) ao HTTP.

Este SSL (antigo TLS) é o protocolo que permite encriptar uma ligação, atribuindo um certificado digital a um *site* que prova que as suas ligações são seguras. É possível ver este certificado no nosso computador quando acedemos a qualquer *site* clicando no aloquete junto à barra de pesquisa do nosso *browser*.

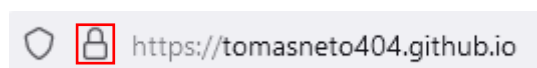


Figura 15 - Aloquete na barra de pesquisa do browser

Após clicar no aloquete, deverá aparecer algo como “Ligação segura”, depois “Mais informação” e finalmente “Ver certificado”.

Neste certificado, há uma parte que diz “Informação da chave pública” e mostra qual o algoritmo utilizado e a chave pública em si.

Informação da chave pública	
Algoritmo	RSA
Tamanho da chave	2048
Exponente	65537
Módulos	B8:B0:60:0E:1A:2F:F1:B1:86:4B:64:EC:11:9F:A6:79:BE:E8:87:F1:88:C5:B4:49:9B:10:B B:CA:AF:EA:AF:BE:54:0C:78:43:7F:CA:7B:4E:45:5B:0B:24:29:F1:BB:23:FC:19:A4:C7:6C: 70:49:76:53:D3:09:23:65:B2:48:7B:B6:1C:AA:07:1A:E2:79:1A:F9:7A:5E:E7:16:F8:A6:4 A:D5:39:A3:E2:0D:F7:57:EF:ED:F8:08:76:5B:52:DA:8B:D0:E6:1E:6E:2F:F9:0F:99:4B:6A: 52:CA:34:E1:A4:C9:20:33:D3:97:E8:7A:77:C5:03:10:26:41:82:61:47:A2:AF:C4:56:3F:7 6:A2:38:CB:B2:70:AE:72:7A:43:C1:7E:27:A3:5E:D6:E3:F6:E7:A5:30:70:BD:2A:96:27:7 A:7B:FB:40:D2:57:77:AF:23:12:27:42:3A:C6:0B:6A:8C:BD:BA:2D:EE:3F:9F:15:EE:62:5 7:A4:A6:95:50:AF:43:B0:AC:76:B8:E1:0E:D9:FF:56:EC:74:50:86:B5:1F:96:2C:D1:95:0 5:E5:B7:05:67:93:4E:9E:F2:5A:38:1F:A7:8F:43:5A:DE:3C:57:DA:48:7A:50:C6:88:38:1 5:C8:97:2C:2C:EC:F8:39:09:36:BD:19:8D:03:56:41:66:07:24:E3

Figura 16 - Informação de chave pública de certificado digital

Para além do RSA também é possível aparecer o *Elliptic Curve*, mas isso é assunto para outro estudo.

Para além dos *sites* que visitamos todos os dias, as nossas aplicações de troca de mensagens que dizem ter criptografia ponto a ponto, também podem utilizar este algoritmo. Desta forma, se dois utilizadores fornecerem as suas chaves públicas um ao outro, podem trocar mensagens encriptadas que nem mesmo a empresa da aplicação as conseguiria desencriptar.

A minha implementação do algoritmo em *python*

De modo a aprofundar o meu conhecimento sobre este algoritmo, desenvolvi um código em *python* com exatamente a mesma lógica que foi explicada neste documento.

O programa permite encriptar com uma chave pública conhecida ou gerar uma cifra toda do zero onde os números primos são escolhidos com um dado número de *bits* à escolha do utilizador e faz todos os cálculos necessários para descobrir o resto das variáveis. No fim mostra todos os dados da chave pública e privada e a mensagem encriptada.

```

Key size (bits): 2048
Message to encrypt: Isto é um teste

p: 319492593438692819728993901879736843310138851440378071644760482213053451161095316086201808100171896031222911020805791760037628756477037300806982572925
9928126612754185837437485043367407117504800537924905344688735348780760029556127434066772233663554762848726720070660784507911741368454052829856465221984469
429

q: 16611387464414877880752933789440462765640597352900857263110302547210484957807436950685875887447455840063299499331697955471321753769717424607486134902
209273169474875987908589710291105498207799632859898924907553586899339097142107705609821404389439309844878693102199874795063198581759078467889741836712725
8143

n: 530721526162090096827512460396328521301425533896543611707371862806165124061745899648542650190045532610133580810738387564743271125247311803965638405703
0149333141526029586401857221391101490994008050091429347525268421184376425138277880605635340726276966643415441642294274629500323900699593056071617885122605
5805972465082696989098631047322682768144611378555414590840827610279394693709272637779054022732252163001314708525997026772696179364186266800212004646580791
27728397587573451860506497087010654914749695189621080603988235843668327557545743382991102591328346111634112164638550083533641193469632637028428797374810
347

e: 65537

d: 3177105701499501162524039299743527606728844465166714301557184535983379817860823180937680057815593696145587542547686027119452717641283973694325097144680
8121348049826673768456048710908907713087827723028015435425162213228048892800251096171641840833636331115570105771581416246382748792226484800476948662389416
4314183692097811308351720949491588325973316350819908466672405512979254068116741610926168232015705419344238062956710263889150754619071170424207025247977698
01702123236700030114674580270170926869613353633969780544401732071431034214939976595960920987497690929887605752833255897167399631129433735550292555881189
657

m: 73115116111322332117109321610115116101

c: 2563909008047446573746006859731536026484338330747545633972360190524294362500731583520330914092599956017094875172778298209885345321984926345886158114812
27707762016406360085731913589422110795015189551128839061336387583756657800648674650327846099413278122533934425828565012276473226347767656981146510831168
4006893246110928016752838447706473612765458089285422579175367351334435234084968024387726607280729459143567300748351208684040447285556846505129813791863964
457472271905981589001195489490915726135126111651665168804116868232344612110211259249659509313781334441642114430313747418996384868749057870319963512987658
248

```

Figura 17 - Cifra RSA implementada em *python*

A implementação não está perfeita pois a mensagem é primeiramente codificada em *ASCII*, o que deveria ser substituída por *UTF-8*.

Para além de encriptar também é possível desencriptar com uma chave privada conhecida.

Esta versão do programa faz com que apenas seja possível utilizar mensagem com letras maiúsculas e não é de todo o código mais eficiente, mas certamente vou melhorar isso em breve. Para já, para testar os conhecimentos é o suficiente.

O programa está disponível na minha página do Github.

Referências

<https://911electronic.com/logic-gates-what-are-logic-gates/xor-gate-truth-table/>

[https://ocw.mit.edu/courses/6-045j-automata-computability-and-complexity-spring-2011/a58157daa3e96833038e8169b8978393 MIT6_045JS11_rsa.pdf](https://ocw.mit.edu/courses/6-045j-automata-computability-and-complexity-spring-2011/a58157daa3e96833038e8169b8978393/MIT6_045JS11_rsa.pdf)

https://www.youtube.com/watch?v=qph77bTKJTM&t=538s&ab_channel=TomRocksMaths

https://www.youtube.com/watch?v=JD72Ry60eP4&t=83s&ab_channel=Computerphile

<https://pt.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/modular-inverses>