RSA

Ranieri S. Althoff[1]

[1]Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Segurança em Computação

1. Introdução

O RSA é um algoritmo de criptografia de chave pública, onde a chave de cifragem é pública e diferente da chave de decifragem, que é privada. No RSA, essa assimetria de chaves é baseada na dificuldade de fatoração do produto de dois números primos grandes.

O nome RSA vem dos criadores Ron Rivest, Adi Shamir e Leonard Adleman e foi publicado em 1977. Como o RSA é um algoritmo lento, ele não é frequentemente utilizado para criptografar mensagens, em vez disso sendo usado para criptografar chaves simétricas de outros algoritmos que são mais rápidos.

2. Operação do RSA

O princípio do RSA é a de que é simples encontrar três inteiros positivos e, d e n tal que para todo m, $(m^e)^d \equiv m \pmod{m}$ e mesmo se sabendo e e n ou até mesmo m, é extremamente custoso encontrar d.

Além disso, para algumas operações é conveniente que a ordem da exponenciação seja invertida, de forma que $(m^d)^e \equiv m \pmod{m}$.

Para que Beto envie suas mensagens criptografadas, Alice transmite sua chave pública (n, e) por uma rota não necessariamente secreta e mantém sua chave privada escondida.

2.1. Cifragem e decifragem

Beto transforma sua mensagem M em um inteiro m tal que $0 \leq m < n$ e computa o texto cifrado c usando a chave pública de Alice, correspondendo a $c \equiv m^e \pmod{n}$.

Alice pode recuperar o texto claro m a partir de c usando sua chave privada calculando $m \equiv c^d \equiv (m^e)^d \pmod{n}$ o que é correto por definição.

2.2. Geração de chaves

Para gerar o par de chaves RSA, utiliza-se o seguinte algoritmo:

1. Escolher dois números primos distintos p e q.
2. Computar $n = pq$. O comprimento de n, expressado em bits, é o tamanho da chave.
3. Computar $\phi(n) = \phi(p)\phi(q) = (p-1)(q-1) = n - (p+q-1)$.
4. Encontrar um inteiro e tal que $1 < e < \phi(n)$ e $gcd(e, \phi(n)) = 1$. Qualquer número primo satisfaz essa condição.
5. Determinar d como $d \equiv e^{-1} \pmod{\phi(n)}$, ou seja, d é a inversa multiplicativa de e módulo $\phi(n)$. De forma simples, encontrar d para que $de \equiv 1 \pmod{\phi(n)}$.

Por razões de segurança, os inteiros p e q devem ser aleatórios, ter magnitude semelhante, mas diferenciar em comprimento por alguns dígitos para dificultar a fatoração [Rivest et al. 1978]. A primalidade pode ser verificada de forma eficiente com um teste probabilístico.

É possível melhorar o desempenho utilizando um e curto e com uma baixa quantidade de bits em 1, geralmente sendo usado $e = 2^{16} + 1 = 65537$. Em geral, números na forma $2^x + 1$ são bons candidatos. No entanto, valores muito baixos de e (por exemplo, 3) podem ser inseguros [Boneh 1999].

Como os fatores comuns de $p - 1$ e $q - 1$ também serão fatores de $pq - 1$, é recomendado que $p - 1$ e $q - 1$ tenham poucos fatores comuns, de preferência nenhum além de 2.

2.3. Exemplo

1. Escolher dois primos distintos, por exemplo $p = 227$ e $q = 281$.
2. Computar $n = pq$, neste exemplo $n = 227 \times 281 = 63787$. O comprimento em bits de n é 16, neste caso.
3. Computar $\phi(n) = \phi(p)\phi(q) = 226 \times 280 = 63280$.
4. Encontrar e relativamente primo a $\phi(n)$, como 257.
5. Encontrar d para que $d \times 257 \equiv 1 \pmod{63280}$. O menor número que satisfaz essa condição é 7633.

A chave pública é $(n = 63787, e = 257)$ e a função de cifragem é $c = m^{257} \pmod{63787}$. A chave privada é $(d = 7633)$ e a função de decifragem é $m = c^{7633} \pmod{63787}$.

Se quisermos criptografar o caractere 'a', por exemplo, correspondente a 97 em ASCII, aplicamos a função de cifragem nesse valor:

$$c = 97^{257} \pmod{63787} = 10888$$

E, para encontrar o texto original, usamos a função de decifragem no texto cifrado:

$$m = 10888^{7633} \pmod{63787} = 97$$

3. Código

A classe `KeyPair` gera um par de chaves como explicado acima recebendo como parâmetro do construtor o tamanho da chave n e define e como 65537. Depois, imprime os valores de p, q, n e da chave pública e privada.

```
import math
import random
import sys

# teste de primalidade de Fermat
def fermat(p, tests=20):
    if p & 1 == 0:
        return False
    for _ in range(tests):
        if pow(random.randrange(2, p-1), p-1, p) != 1:
            return False
```

```
12      return True
13
14  # algoritmo de Euclides extendido e adaptado
15  def inv_mul(a, b):
16      x, y, newx, newy = 0, b, 1, a
17      while newy:
18          q = y // newy  # quociente
19          x, y, newx, newy = newx, newy, x - q*newx, y - q*newy
20      assert(x <= 1)  # se x > 1, a nao tem inversa multiplicativa
21      return x % b
22
23  # gera um primo aleatorio no intervalo [lower, upper)
24  def random_prime(lower, upper):
25      while True:
26          p = random.randrange(lower, upper)
27          if fermat(p):
28              return p
29
30
31  class KeyPair:
32      def __init__(self, b):
33          # o tamanho da chave precisa ser potencia de 2 e positivo
34          assert(b & (b-1) == 0 and b > 0)
35
36          # tamanho em bits de p e q para que n tenha b bits
37          l = b // 2
38
39          # p e q nao sao necessarios apos gerar n e phi, mas sao
40          # guardados para fins didaticos
41          p = self.p = random_prime(2 ** (l-1), 2 ** l)
42          q = self.q = random_prime(2 ** (l-1), 2 ** l)
43          self.n = p * q
44
45          # e pode ser qualquer primo, 65537 eh rapido e razoavel
46          self.e = 65537
47
48          # encontrar inversa multiplicativa pelo algoritmo de Euclides
49          self.d = inv_mul(self.e, self.n - (p+q-1))
50          assert((self.d * self.e) % (self.n - (p+1-1)) == 1)
51
52      def public_key(self):
53          return self.n, self.e
54
55      def private_key(self):
56          return self.d
57
58  # cria um par de chaves de tamanho indicado pelo primeiro argumento
59  kp = KeyPair(int(sys.argv[1]))
60  print('p = {}\nq = {}\nn = {}'.format(kp.p, kp.q, kp.n))
61  print('public key = {}'.format(kp.public_key()))
62  print('private key = {}'.format(kp.private_key()))
```

A saída do algoritmo usando diferentes tamanhos de chave como parâmetro pode ser vista nas figuras anexas ao fim do documento.

Anexo - Questionário

- **O que é a função totiente de Euler?**
  A função totiente de Euler, ou $\phi(n)$, é uma função que conta a quantidade de inteiros positivos até n que sejam relativamente primos a n, ou, mais formalmente, a quantidade de números inteiros k no intervalo $1 \leq k \leq n$ onde $gcd(n, k) = 1$.

- **Por que os expoentes são determinados módulo $\phi(n)$ e não módulo n?**
  O sistema RSA utiliza módulo $\phi(m)$ para determinar e e d com base na propriedade $a^{\phi(n)} \equiv 1 \pmod{n}$, o que significa $a^{\phi(n)+1} \equiv a \pmod{n}$ e é chamado de teorema de Euler.
  Para cifrar e decifrar usando e e d, precisamos que $a^{ed} \equiv a \pmod{n}$, ou seja, precisamos que $ed \equiv 1 \pmod{\phi(n)}$, o que só é verdade quando ambos e e d são relativamente primos a $\phi(n)$.
  Se essa relação não se manter, é impossível cifrar e decifrar um valor unicamente utilizando RSA, porque múltiplos valores m serão cifrados para um mesmo valor c e não é possível determinar o valor original na decifragem.

- **O que é PKCS #1?** É o primeiro padrão de uma família chamada *Public-Key Cryptography Standards*. Esse padrão contém algumas definições e recomendações para a implementação do algoritmo RSA, como as propriedades matemáticas de chaves privadas e públicas, operações primitivas para cifragem e assinatura, entre outros.

- **Como o RSA pode ser usado para ciframento de dados? E como seria usado para assinatura?** Para cifrar dados, é necessário usar a chave pública na mensagem em texto claro e enviar para o dono da chave privada. Dessa forma, apenas quem tem acesso a chave privada pode ler a mensagem cifrada.
  Para assinatura, o processo é inverso, onde o dono da chave privada cifra com a chave privada de forma que qualquer pessoa pode decifrar a mensagem com a chave pública e garantir que quem a cifrou foi o detentor da chave privada.

- **Por que é importante ter bons geradores de números aleatórios para gerar chaves RSA?** O que um bom gerador de números aleatórios tem como característica é de gerar números aleatórios desconhecidos e aparentemente não previsíveis para um potencial atacante.
  Números aleatórios obtidos por um processo físico, como a leitura da variação da velocidade do disco, a temporização da entrada de teclas digitadas, etc. Uma alternativa é usar um gerador pseudo-aleatório alimentado por uma semente aleatória, mas um gerador desse tipo é necessáriamente periódico, enquanto números realmente aleatórios não são [Rivest et al. 2009a].
  Um estudo conduzido em 2012 [Lenstra et al. 2012] utilizou chaves públicas disponíveis na internet e concluiu que cerca de 0.2% de todas as chaves não possuem segurança por utilizarem números aleatórios de baixa qualidade.

- **Qual seria o tamanho da chave desejável RSA nos dias atuais?** A escolha do tamanho de chave depende da importância do dado a ser cifrado. Dito isto,

o maior tamanho de chave já quebrado foi de 768 bits, portanto um tamanho de chave maior do que esse é desejável.
Ainda assim, essa quebra levou mais de dois anos em um supercomputador e foi estimado em mais de 1500 anos em um computador comum [Timmer 2010].
A recomendação tradicional do RSA Laboratories é de se usar ao menos 1024 bits de chave, ou usar 2048 para maior *future-proofing*. Outra recomendação é a de Lenstra-Verheul que recomenda 952 bits no mínimo ou 1369 para *future-proofing* [Rivest et al. 2009b].

Referências

Boneh, D. (1999). Twenty years of attacks on the rsa cryptosystem. *Notices of the American Mathematical Society*, pages 203–213.

Lenstra, A. K., Hughes, J. P., Augier, M., Bos, J. W., Kleinjung, T., and Wachter, C. (2012). Ron was wrong, whit is right. Cryptology ePrint Archive, Report 2012/064. Disponível em: `http://eprint.iacr.org/2012/064.pdf`. Acesso em: 19 mai 2016.

Rivest, R. et al. (2009a). How does one find random numbers for keys? RSA Laboratories. Disponível em: `http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/find-random-numbers-for-keys.htm`. Acesso em: 19 mai 2016.

Rivest, R. et al. (2009b). What key size should be used? RSA Laboratories. Disponível em: `http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/key-size.htm`. Acesso em: 19 mai 2016.

Rivest, R., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, pages 120–126.

Timmer, J. (2010). 768-bit rsa cracked, 1024-bit safe (for now). Ars Technica. Disponível em: `http://arstechnica.com/security/2010/01/768-bit-rsa-cracked-1024-bit-safe-for-now/`. Acesso em: 19 mai 2016.

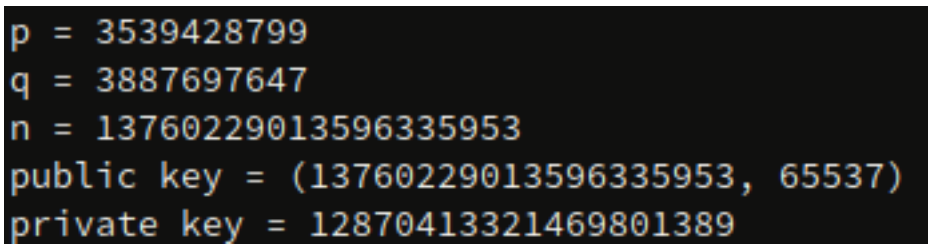

Figura 1. Saída do algoritmo para 64 bits

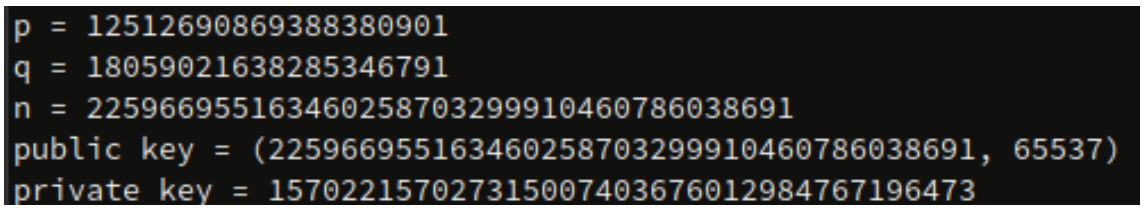

Figura 2. Saída do algoritmo para 128 bits

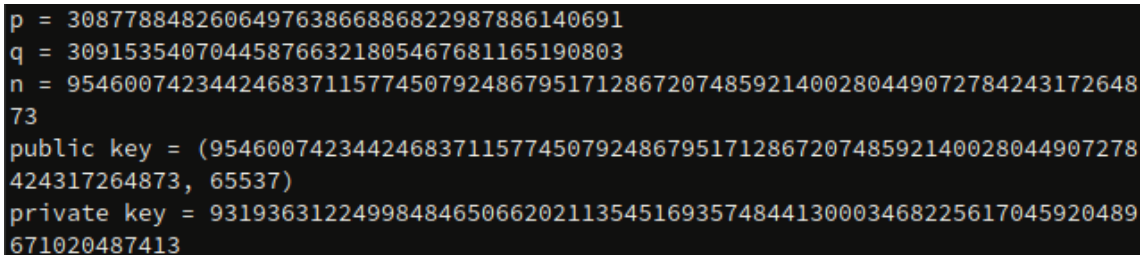

Figura 3. Saída do algoritmo para 256 bits

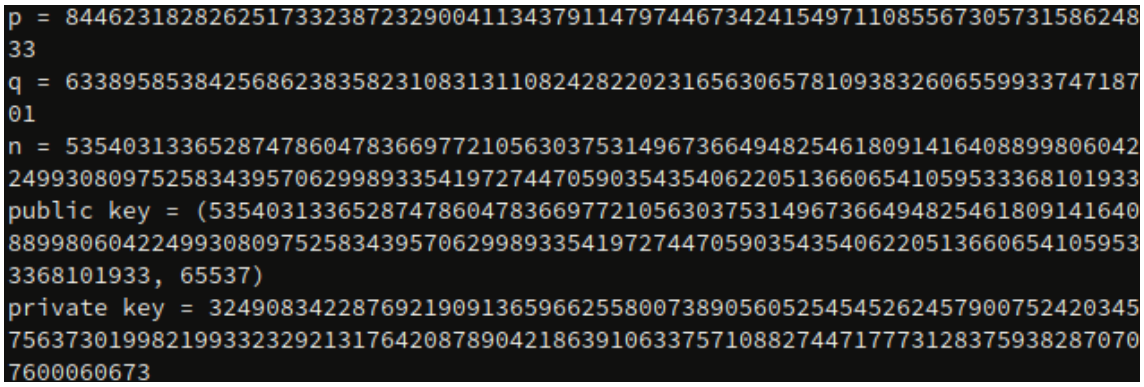

Figura 4. Saída do algoritmo para 512 bits

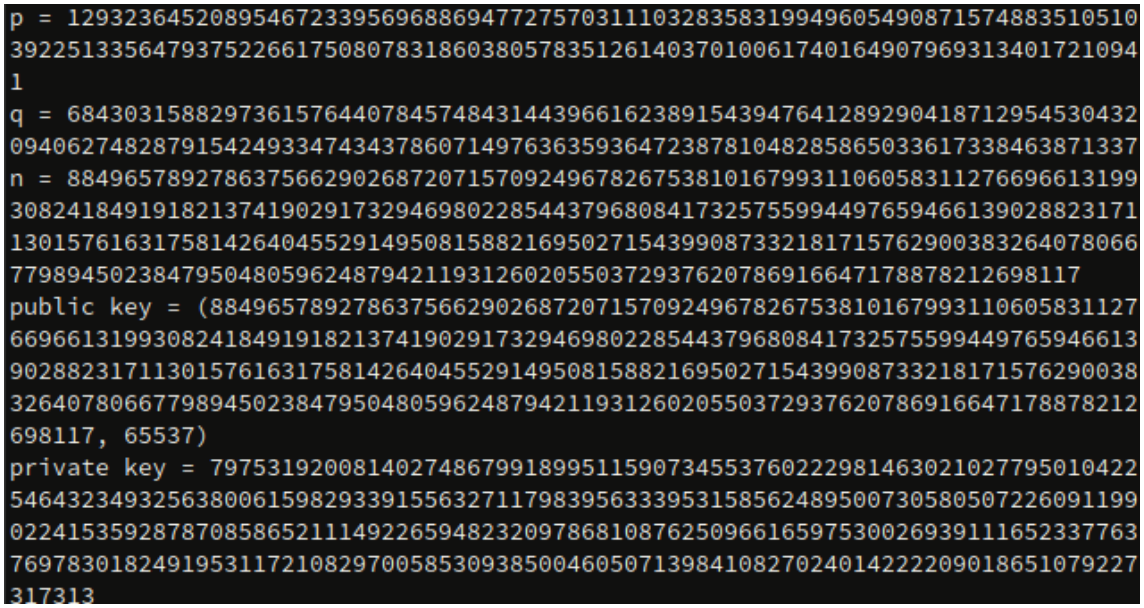

Figura 5. Saída do algoritmo para 1024 bits

Figura 6. Saída do algoritmo para 2048 bits

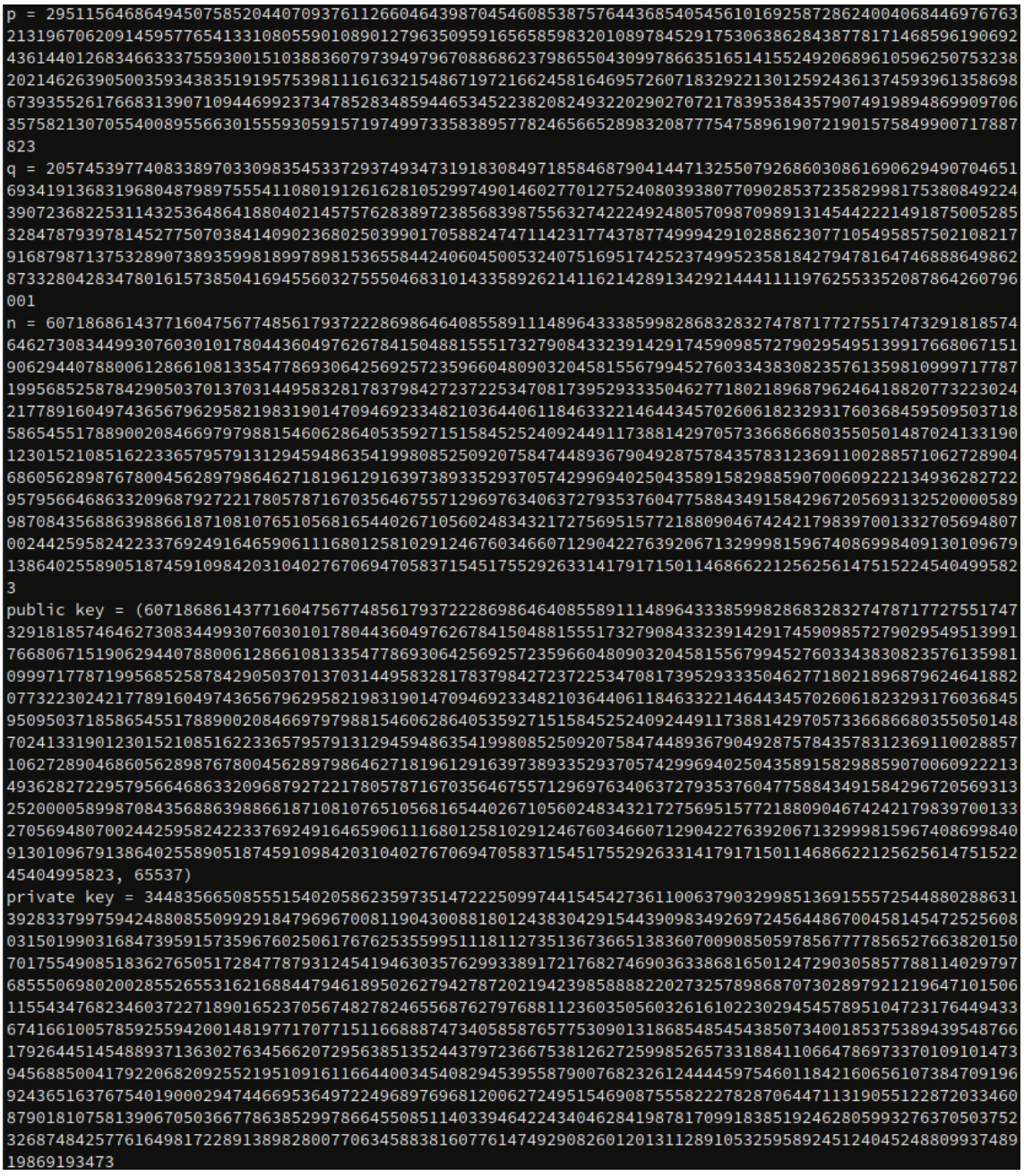

Figura 7. Saída do algoritmo para 4096 bits