RSA

Ranieri S. Althoff¹

¹Universidade 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 $0 \le m < n$ 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 $p \in q$.
- 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 $n \notin 16$, 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
```

```
return True
12
   # algoritmo de Euclides extendido e adaptado
14
  def inv_mul(a, b):
       x, y, newx, newy = 0, b, 1, a
17
       while newy:
           q = y // newy # quociente
18
           x, y, newx, newy = newx, newy, x - q*newx, y - q*newy
19
       assert(x \le 1) # se x > 1, a nao tem inversa multiplicativa
       return x % b
21
22
  # gera um primo aleatorio no intervalo [lower, upper)
23
  def random_prime(lower, upper):
       while True:
25
          p = random.randrange(lower, upper)
26
27
           if fermat(p):
               return p
29
30
  class KeyPair:
31
       def ___init_
                  _(self, b):
           # o tamanho da chave precisa ser potencia de 2 e positivo
33
           assert (b & (b-1) == 0 and b > 0)
34
35
           # tamanho em bits de p e q para que n tenha b bits
           1 = b // 2
37
38
           # p e q nao sao necessarios apos gerar n e phi, mas sao
           # guardados para fins didaticos
40
           p = self.p = random\_prime(2 ** (1-1), 2 ** 1)
41
           q = self.q = random_prime(2 ** (1-1), 2 ** 1)
42
           self.n = p * q
43
           # e pode ser qualquer primo, 65537 eh rapido e razoavel
45
           self.e = 65537
46
           # encontrar inversa multiplicativa pelo algoritmo de Euclides
48
           self.d = inv_mul(self.e, self.n - (p+q-1))
49
           assert((self.d * self.e) % (self.n - (p+1-1)) == 1)
50
51
      def public key(self):
52
          return self.n, self.e
53
54
       def private_key(self):
           return self.d
56
57
  # 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 $1 \le k \le n$ 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.

```
p = 3539428799
q = 3887697647
n = 13760229013596335953
public key = (13760229013596335953, 65537)
private key = 12870413321469801389
```

Figura 1. Saída do algoritmo para 64 bits

```
p = 12512690869388380901
q = 18059021638285346791
n = 225966955163460258703299910460786038691
public key = (225966955163460258703299910460786038691, 65537)
private key = 157022157027315007403676012984767196473
```

Figura 2. Saída do algoritmo para 128 bits

```
p = 308778848260649763866886822987886140691
q = 309153540704458766321805467681165190803
n = 954600742344246837115774507924867951712867207485921400280449072784243172648
73
public key = (95460074234424683711577450792486795171286720748592140028044907278
424317264873, 65537)
private key = 93193631224998484650662021135451693574844130003468225617045920489
671020487413
```

Figura 3. Saída do algoritmo para 256 bits

```
p = 844623182826251733238723290041134379114797446734241549711085567305731586248
33
q = 633895853842568623835823108313110824282202316563065781093832606559933747187
01
n = 535403133652874786047836697721056303753149673664948254618091416408899806042
2499308097525834395706299893354197274470590354354062205136606541059533368101933
public key = (53540313365287478604783669772105630375314967366494825461809141640
8899806042249930809752583439570629989335419727447059035435406220513660654105953
3368101933, 65537)
private key = 32490834228769219091365966255800738905605254545262457900752420345
7563730199821993323292131764208789042186391063375710882744717773128375938287070
7600060673
```

Figura 4. Saída do algoritmo para 512 bits

```
p = 129323645208954672339569688694772757031110328358319949605490871574883510510
3922513356479375226617508078318603805783512614037010061740164907969313401721094
q = 684303158829736157644078457484314439661623891543947641289290418712954530432
0940627482879154249334743437860714976363593647238781048285865033617338463871337
n = 884965789278637566290268720715709249678267538101679931106058311276696613199
3082418491918213741902917329469802285443796808417325755994497659466139028823171
1301576163175814264045529149508158821695027154399087332181715762900383264078066
779894502384795048059624879421193126020550372937620786916647178878212698117
public key = (88496578927863756629026872071570924967826753810167993110605831127
6696613199308241849191821374190291732946980228544379680841732575599449765946613
9028823171130157616317581426404552914950815882169502715439908733218171576290038
3264078066779894502384795048059624879421193126020550372937620786916647178878212
698117, 65537)
private key = 79753192008140274867991899511590734553760222981463021027795010422
5464323493256380061598293391556327117983956333953158562489500730580507226091199
0224153592878708586521114922659482320978681087625096616597530026939111652337763
7697830182491953117210829700585309385004605071398410827024014222209018651079227
```

Figura 5. Saída do algoritmo para 1024 bits

```
p = 946588692303918496833027280577191419072647495072321125835483570016379259561
3064890510842248999955136787585493544580024364159976440445087735285634058868368
9979131002221083180128753630898619112795841234178617362891197193571819549881699
005038023288601871964515079691740314306965043967917411570047638519272847587
q = 119999815218654956029443591783674346848095630286077363298153924171238045459
9380203659812063608412035895850333292941463551821342295867918129190051426025999
8965203167162382270004852506736838171058634908268977737197983202199436083658867
0657942695485534220652018663604451263372841328189387390110102029600258834037
n = 113590468164538452288723168155066481982863360715047545200183769071336186052
0924458320731610107542409238301989540963757917855212459587879772898750011126934
2414609783107013925002588883159253673843990403864506935754362274010833200944271
2212242737813589862789303092674854409089469945267329065165790319087681868514660
7374857081320545934465310646632845886353535126732855448833087272847664075146240
0782828427366742292298476620459574142903831575764804080558416251168243306576015
4938120232504165384098853084802519104638585713552826062056783343599962129592146
21607786171601446246394083823096090970148073104060422668645428918719
public key = (11359046816453845228872316815506648198286336071504754520018376907
1336186052092445832073161010754240923830198954096375791785521245958787977289875
0011126934241460978310701392500258888315925367384399040386450693575436227401083
3200944271221224273781358986278930309267485440908946994526732906516579031908768
1868514660737485708132054593446531064663284588635353512673285544883308727284766
4075146240078282842736674229229847662045957414290383157576480408055841625116824
3306576015493812023250416538409885308480251910463858571355282606205678334359996
212959214621607786171601446246394083823096090970148073104060422668645428918719,
65537)
private key = 87392748917288826789477693283408794338159152447840568290945054001
7839262266903474944656136912621928965525778058722166131102972712167784212110727
8798395147523222828079128677334643585557103449083139053414959011163258229399792
9666907555047770012756714955850317233606230206068469011097078995434479446252332
1966848953797645830150003849704990908193422845165493604148568697404821651834869
5224923252290927400515637478022809711645254742017588236693570233929363024107515
7286695053006451621533210850844441812819427961190096996437797356454435666517100
27475818329320510616637582712531881575268625387184193956736166215000942833649
```

Figura 6. Saída do algoritmo para 2048 bits

```
295115646864945075852044070937611266046439870454608538757644368540545610169258728624004068446976763
2131967062091459577654133108055901089012796350959165658598320108978452917530638628438778171468596190692
2021462639050035934383519195753981116163215486719721662458164695726071832922130125924361374593961358698
6739355261766831390710944699237347852834859446534522382082493220290270721783953843579074919894869909706
q = 205745397740833897033098354533729374934731918308497185846879041447132550792686030861690629490704651
6934191368319680487989755541108019126162810529974901460277012752408039380770902853723582998175380849224
3907236822531143253648641880402145757628389723856839875563274222492480570987098913145442221491875005285
3284787939781452775070384140902368025039901705882474711423177437877499942910288623077105495857502108217
9168798713753289073893599818997898153655844240604500532407516951742523749952358184279478164746888649862
8733280428347801615738504169455603275550468310143358926214116214289134292144411119762553352087864260796
001
n = 607186861437716047567748561793722286986464085589111489643338599828683283274787177275517473291818574
6462730834499307603010178044360497626784150488155517327908433239142917459098572790295495139917668067151
9062944078800612866108133547786930642569257235966048090320458155679945276033438308235761359810999717787
1995685258784290503701370314495832817837984272372253470817395293335046277180218968796246418820773223024
2177891604974365679629582198319014709469233482103644061184633221464434570260618232931760368459509503718
5865455178890020846697979881546062864053592715158452524092449117388142970573366866803550501487024133190
1230152108516223365795791312945948635419980852509207584744893679049287578435783123691100288571062728904
6860562898767800456289798646271819612916397389335293705742996940250435891582988590700609222134936282722
9579566468633209687927221780578716703564675571296976340637279353760477588434915842967205693132520000589
9870843568863988661871081076510568165440267105602483432172756951577218809046742421798397001332705694807
0024425958242233769249164659061116801258102912467603466071290422763920671329998159674086998409130109679
1386402558905187459109842031040276706947058371545175529263314179171501146866221256256147515224540499582
public key = (60718686143771604756774856179372228698646408558911148964333859982868328327478717727551747
3291818574646273083449930760301017804436049762678415048815551732790843323914291745909857279029549513991
7668067151906294407880061286610813354778693064256925723596604809032045815567994527603343830823576135981
0999717787199568525878429050370137031449583281783798427237225347081739529333504627718021896879624641882
0773223024217789160497436567962958219831901470946923348210364406118463322146443457026061823293176036845
9509503718586545517889002084669797988154606286405359271515845252409244911738814297057336686680355050148
7024133190123015210851622336579579131294594863541998085250920758474489367904928757843578312369110028857
1062728904686056289876780045628979864627181961291639738933529370574299694025043589158298859070060922213
4936282722957956646863320968792722178057871670356467557129697634063727935376047758843491584296720569313
2520000589987084356886398866187108107651056816544026710560248343217275695157721880904674242179839700133
9130109679138640255890518745910984203104027670694705837154517552926331417917150114686622125625614751522
45404995823, 65537)
private key = 34483566508555154020586235973514722250997441545427361100637903299851369155572544880288631
3928337997594248808550992918479696700811904300881801243830429154439098349269724564486700458145472525608
0315019903168473959157359676025061767625355995111811273513673665138360700908505978567777856527663820150
7017554908518362765051728477879312454194630357629933891721768274690363386816501247290305857788114029797
1155434768234603722718901652370567482782465568762797688112360350560326161022302945457895104723176449433
6741661005785925594200148197717077151166888747340585876577530901318685485454385073400185375389439548766
9456885004179220682092552195109161166440034540829453955879007682326124444597546011842160656107384709196
9243651637675401900029474466953649722496897696812006272495154690875558222782870644711319055122872033460
8790181075813906705036677863852997866455085114033946422434046284198781709918385192462805993276370503752
3268748425776164981722891389828007706345883816077614749290826012013112891053259589245124045248809937489
19869193473
```

Figura 7. Saída do algoritmo para 4096 bits