Fundação Getúlio Vargas
Escola de Matemática Aplicada

Disciplina de Álgebra e Criptografia

ATAQUES AO RSA - Explorando vulnerabilidades do sistema e o problema da fatoração

Nicole dos Santos de Souza
Wendell Oliveira

Rio de Janeiro
2023

Sumário

1 Introdução

O universo da criptografia, ao longo da história, tem sido palco de uma constante batalha entre a criação de sistemas robustos e os esforços incansáveis dos criptoanalistas em busca de suas vulnerabilidades. O ditado "Uma corrente não é mais forte que seu elo mais fraco"ressoa de maneira significativa nesse contexto, onde os atacantes instintivamente direcionam seus esforços ao ponto mais suscetível da defesa, muitas vezes descobrindo fragilidades que podem ter escapado à percepção dos próprios projetistas.

O RSA (Rivest-Shamir-Adleman), um dos pilares da criptografia assimétrica, tem resistido bem a ataques e é uma técnica amplamente adotada globalmente. No entanto, como em qualquer sistema, a aparente invulnerabilidade pode revelar-se como um desafio. Este trabalho se propõe a explorar a complexidade dos ataques direcionados ao RSA, explorando suas vulnerabilidades, ou seja, seus elos mais fracos, e também o ataque mais discutido: a fatoração da chave pública.

A fatoração da chave pública no contexto do RSA representa a tentativa de desvendar a força de seu mecanismo de segurança. Se bem-sucedido, tal ataque permitiria a decodificação de todas as mensagens criptografadas com a chave pública. Contudo, a resistência do RSA reside na dificuldade computacional envolvida na fatoração de números primos extremamente grandes, destacando a essencialidade de compreender a complexidade inerente a esse processo.

Em resumo, neste trabalho, foram estudados diferentes tipos de ataques ao RSA, compreendendo a matemática por trás de cada um e explorando suas implementações com situações simuladas do mundo real, para que se pudesse entender melhor como o RSA pode ser vulnerável em certas situações.

2 Tipos de Ataques ao RSA

A segurança do sistema de Criptografia RSA se baseia na complexidade computacional associada à fatoração do produto de dois números primos muito grandes, que é praticamente impossível do ponto de vista prático. Mas será que a segurança do RSA baseia-se apenas nesse fato? Chamamos de "ataque"a tentativa de recuperar a chave privada do sistema. Nesta seção, vamos explorar algumas maneiras de se (tentar) fazer isso. Os ataques mais comuns ao RSA são: a Força Bruta, Ataques Matemáticos e Ataques Temporais.

2.1 Força bruta

O método de ataque conhecido como força bruta consiste, de modo geral, na tentativa de explorar todas as possíveis combinações de chaves até encontrar aquela capaz de decifrar a mensagem. Quando a chave é de dimensões consideráveis, é necessário muito poder de processamento para testar todas as combinações em um intervalo de tempo razoável. Portanto, na maioria dos cenários, esse tipo de ataque é considerado impraticável. A estratégia de defesa contra ataques de força bruta reside precisamente na geração de chaves dentro de um intervalo tão extenso que se torna computacionalmente muito caro, em termos de tempo e/ou operações, determinar qual chave deve ser empregada.

2.2 Ataques Matemáticos

Ataques matemáticos referem-se a tentativa de quebrar o sistema utilizando ferramentas matemáticas. Há algumas possibilidades:

1. Fatorar N para que possamos calcular $\varphi(N) = (p-1) \cdot (q-1)$ e em seguida $d \equiv e^{-1} \pmod{\varphi(N)}$.

2. Tentar determinar $\varphi(N)$ sem precisar achar p e q que compõem N, e com ele determinar d.

3. Encontrar d diretamente sem calcular $\phi(N)$.

Entretanto, esses métodos são inviáveis na prática, pois exigem muito poder computacional devido ao enorme tamanho de n. E muitos dessas tentativas apesar de tentar evitar o problema que é fatorar N acabam se mostrando equivalentes a ele.

2.3 Ataques Temporais

É preciso considerar também os ataques temporais, uma categoria que se destaca por não depender de falhas ou artifícios matemáticos, mas sim de questões relacionadas à implementação computacional do sistema RSA. Essas abordagens buscam identificar vulnerabilidades que possam surgir durante a execução do algoritmo, explorando aspectos temporais e operacionais. Ao analisar a segurança do RSA, é imperativo não apenas compreender as possíveis falhas algorítmicas, mas também avaliar a robustez da implementação prática, reconhecendo que ameaças podem surgir não apenas das equações matemáticas, mas também da execução real do sistema em ambientes computacionais específicos.

3 Exemplos de Ataques ao RSA

3.1 Algoritmo de fatorização de Fermat

Como discutido, a força do RSA reside na dificuldade que é fatorar números grandes de forma eficiente, entretanto no caso do RSA temos a informação de que o N é composto por exatamente 2 primos, usando isso, e uma manipulação algébrica simples é possível chegar no algoritmo de fatorização de Fermat.

Dado $N = p \cdot q$, podemos calcular $a = \frac{p+q}{2}$ e $b = \frac{p-q}{2}$, eles satisfazem $p = a + b$ e $q = a - b$, e daí podemos substituir:

$$N = (a+b) \cdot (a-b) = a^2 - b^2$$

Com isso, fatorar N se resume a encontrar a e b tais que $b^2 = a^2 - N$

Para encontrar esses valores começamos com um chute pra a, sendo esse chute $\lceil \sqrt{N} \rceil$, checamos se $a^2 - N$ é um quadrado perfeito, se isso for nosso trabalho está feito, caso não seja, prosseguimos incrementando o a de uma unidade e repetindo esse processo até encontrar, quando finalmente encontramos podemos calcular p e q e fatorar nosso N.

Estamos tentando encontrar os fatores de N de uma maneira que parece ingênua(testando as possibilidade), porém, essa maneira acaba sendo muito eficiente quando p e q estão

relativamente próximos, já que começamos com a no "centro"e vamos calculando o b até que ambos sejam quadrados perfeitos, como os oincrementos são de 1 em 1 esse é um algoritmo de fatorização que com certeza encontrará a resposta, mas é muito mais inteligente que, por exemplo, calcular N/i para todo i natural até que ambos sejam inteiros, e mais eficiente quando p e q são proximos.

3.1.1 Implementação

```
from math import ceil, sqrt
def Fermat_Factor(N):
    a = ceil(sqrt(N))
    while True:
        b2 = a*a - N
        b = sqrt(b2)
        if b.is_integer():
            break
        a += 1
    return int(a-b), int(a+b)
```

Usando a função com $p = 4441$ e $q = 7919$: $N = 4441 \cdot 7919 = 35168279$

```
Fermat_Factor(35168279)
```

Obtemos a saída:

```
(4441, 7919)
```

Como vemos é um algoritmo bem simples de ser implementado, e eficiente dadas as circunstâncias, esse método não é bem uma ameaça ao RSA mas é algo que deve ser levado em conta ao gerar p e q.

3.2 Algoritmo rho de Pollard

Esse é um algoritmo de fatorização de $N = p \cdot q$, e se baseia naquela mesma idéia do algoritmo de Fermat que é encontrar x e y tais que $x^2 \equiv y^2 \pmod{N}$, já que encontrando eles teremos

$$x^2 - y^2 \equiv 0 \pmod{N}$$

$$(x+y)(x-y) \equiv 0 \pmod{N} \Rightarrow N \mid (x+y)(x-y)$$

Com isso, devemos ter:

$$\text{mdc}(x-y, N) = 1, p, q, \text{ou } N$$

Para encontrar esse x e y, utilizamos o algoritmo de Floyd para detecção de ciclos(também chamado de algoritmo da Tartaruga e da Lebre), dado x_0 definimos indutivamente a

sequência $x_i = h(x_{i-1})$, $h(x) = a \cdot x^2 + b \pmod N$ e queremos encontrar $i \neq j$ tais que $x_i = x_j$, afinal, se

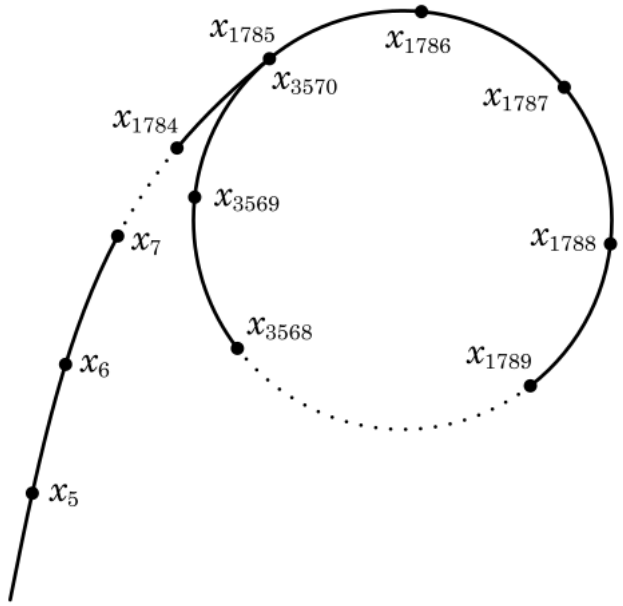
$$x_i = x_j$$
$$h(x_i) = h(x_j)$$
$$a \cdot x_i^2 + b \equiv a \cdot x_j^2 + b \pmod N$$
$$x_i^2 \equiv x_j^2 \pmod N$$

Com isso podemos olhar para o $\text{mdc}(x_i - y_i, N)$ e caso ele não seja 1, encontramos algum dos nossos fatores.

O funcionamento do algoritmo se baseia em construir duas sequências (x_i) e y_i que começam num mesmo ponto, porém, a sequência (y_i)(lebre) tem o dobro da velocidade da (x_i)(tartaruga),

$$x_0 = y_0$$
$$x_i = h(x_{i-1})$$
$$y_i = h(h(y_{i-1}))$$

Computamos cada termo da sequência e os comparamos até que haja uma colisão. Isso vai ocorrer eventualmente já que em algum momento entramos em um ciclo e como a lebre é duas vezes mais rápida que a tartaruga ela tem que passar por ela.



3.2.1 Exemplo:

Fatorando $N = 8051$, utilizaremos $P(x) = x^2 + 1$ e começaremos com $x_0 = y_0 = 2$:

i	x_i	y_i	$\text{mdc}(x_i - y_i, N\}$
0	2	2	
1	5	26	1
2	26	7474	1
3	677	871	97

Como chegamos em um mdc que é diferente de 1, e também é diferente de N, temos que ele tem que ser um dos divisores de N e facilmente verificamos que $N = 8051 = 97 \cdot 83$, caso nosso processo chegasse em 8051 tudo o que nos restaria seria testar outro valor inicial, eventualmente chegaríamos em um dos fatores de N.

3.2.2 Implementação:

```
import random
import math

# Function to calculate (base^exponent)%modulus
def modular_pow(base, exponent,modulus):

    # initialize result
    result = 1

    while (exponent > 0):

        # if y is odd, multiply base with result
        if (exponent & 1):
            result = (result * base) % modulus

        # exponent = exponent/2
        exponent = exponent >> 1

        # base = base * base
        base = (base * base) % modulus

    return result

# method to return prime divisor for n
def PollardRho( n):

    if (n == 1):
        return n

    if (n % 2 == 0):
        return 2

    x = (random.randint(0, 2) % (n - 2))
```

```
    y = x

    c = (random.randint(0, 1) % (n - 1))

    d = 1

    while (d == 1):
        x = (modular_pow(x, 2, n) + c + n)%n
        y = (modular_pow(y, 2, n) + c + n)%n
        y = (modular_pow(y, 2, n) + c + n)%n
        d = math.gcd(abs(x - y), n)
        if (d == n):
            return PollardRho(n)

    return d
```

3.3 Módulo Comum

O ataque que será explorado aqui é considerado elementar pois ilustra o uso inadequado do sistema RSA. Em alguns cenários, a tentativa de simplificar a administração e a gestão de chaves levou à proposta de empregar um módulo único para múltiplos usuários, emitido por uma autoridade central confiável. A ideia por trás dessa abordagem é aparentemente eficiente, permitindo a reutilização do mesmo módulo para todos os usuários. No entanto, tal prática apresenta uma vulnerabilidade significativa, pois permite que um usuário seja capaz de explorar seus próprios expoentes para fatorar o módulo de outros usuários.

Essa situação ressalta a importância crucial de manter a singularidade dos módulos RSA para garantir a robustez e confidencialidade do sistema.

3.3.1 Justificativa Matemática

Relembremos que a criptografia RSA é empregada em uma mensagem M da seguinte forma:

$$C = M^e \bmod n$$

Pelo Teorema de Bezout, sabe-se que dados dois inteiros a e b (não são ambos 0) existem inteiros x e y tais que:

$$ax + by = mdc(a, b)$$

Tem-se, ainda, que o inverso multiplicativo de um número inteiro a no mod n é x , onde x é dado pela seguinte equação:

$$ax \equiv 1 \bmod n$$

É importante relembrar também que, por Bezout, se o denominador a e o módulo n não são coprimos, então o denominador não é invertível mod n .

Com esses conceitos em mente, é possível mostrar como o ataque funciona. No contexto, se as chaves e_1 e e_2 são coprimas, ou seja, se $\text{mdc}(e_1, e_2) = 1$, existirão x e y tais que:

$$xe_1 + ye_2 = 1$$

É possível determinar x e y usando o Algoritmo de Euclides Estendido.[1] Portanto, a mensagem pode ser recuperada da seguinte maneira:

$$C_1^x \times C_2^y = (M^{e_1})^x \times (M^{e_2})^y$$

$$= M^{e_1 x + e_2 y} = M$$

É necessário se atentar ao fato de que normalmente y será um número inteiro negativo. Como tal, C_2^y deve ser tratado da seguinte forma:

$$C_2^y = (\frac{1}{C_2})^{-y}$$

E com isso, tem-se que $\text{mdc}(C_2, n) = 1$.

Conclui-se, então, que é possível recuperar a mensagem M com:

$$M = C_1^x \times (C_2^{-1})^y$$

3.3.2 Implementação em Python

O procedimento descrito acima foi traduzido para código na linguagem de programação Python. Abaixo, apresentamos uma visão geral da implementação.

Funções utilizadas no processo:

```python
import rsa
from math import gcd
from utils import fatorar_em_primos, modinv

def str_to_int(s):
    """Converte uma string para um número inteiro."""
    return int.from_bytes(s.encode(), 'big')

def int_to_str(i):
    """Converte um número inteiro para uma string."""
    return i.to_bytes((i.bit_length() + 7) // 8, 'big').decode()


def encrypt(msg, e, n):
    """Realiza a operação de encriptação usando o algoritmo RSA."""
    return pow(msg, e, n)

```

[1] https://pt.wikipedia.org/wiki/Algoritmo_de_Euclides_estendido

```python
# definindo exceção para quando p e q são descobertos, ela retorna p e q
class PAndQFoundException(Exception):
    def __init__(self, p, q):
        self.p = p
        self.q = q

def attack(c1, c2, e1, e2, N):
    """Função de ataque ao RSA."""
    # print("Iniciando o ataque...")

    # Verifica se os expoentes são coprimos
    if gcd(e1, e2) != 1:
        raise ValueError("Expoentes e1 e e2 precisam ser coprimos.")

    # Calcula o inverso modular de e1 modulo e2
    x = modinv(e1, e2)

    # Calcula o coeficiente y usando o algoritmo extendido de Euclides
    y = (gcd(e1, e2) - e1 * x) // e2

    # Calcula o inverso modular de c2 modulo N
    try:
        temp = modinv(c2, N)
    except:
        print(f"Não foi possível encontrar o inverso modular de {c2} modulo {N}.")
        print("Logo, eles possuem um fator em comum.")

        # Fatorando o c2
        fatores = fatorar_em_primos(c2)
        for fator in fatores:
            if N % fator == 0:
                p = fator
        q = N // p

        # levanta um erro especial, indicando que p e q foram descobertos
        raise PAndQFoundException(p, q)

    # Calcula os termos do produto que resulta na mensagem original
    m1 = pow(c1, x, N)
    m2 = pow(temp, -y, N)

    # Retorna a mensagem descriptografada
    return (m1 * m2) % N


```

A seguinte função gera uma chave pública aleatória (e_1, n) para que se possa simular um exemplo de uso.

```python
import rsa

def generate_keys(size):
    # Gera as chaves n, e1, e2 para teste
    pub, priv = rsa.newkeys(size)
    return pub.n, pub.e
```

Por fim, temos um exemplo de uso , onde uma mensagem foi criptografada e depois descriptografada utilizando o ataque.

```python
# Primeiro Exemplo de uso
from common_module_functions import *

n, e1 = generate_keys(512)
e2 = 587

msg = "minha senha do banco é 1234"
msg_int = str_to_int(msg)
if msg_int > n:
    raise ValueError("A mensagem deve ser menor que n.")
else:
    c1 = encrypt(msg_int, e1, n)
    c2 = encrypt(msg_int, e2, n)
    print(f"Mensagem criptografada: {c1}")
    print(f"Expoente e1: {e1}")
    print(f"Expoente e2: {e2}")

    # Realiza o ataque
    try:
        message = attack(c1, c2, e1, e2, n)
        message = int_to_str(message)
        print("Ataque bem sucedido!")
        print(f"Mensagem descriptografada: {message}")
    except Exception as e:
        print("Ataque falhou.")
        print(e)
```

Saída:

```
Mensagem criptografada:
1106529244324875623119541421093000040567409498197705559280987017561560994267560 58
8663727444864989157243141602198201885183731947252722701113410811445254005

Expoente e1: 65537
Expoente e2: 587
Ataque bem sucedido!
Mensagem descriptografada: minha senha do banco é 1234
```

3.4 Ataque de Hastad

Outro ponto de vulnerabilidade do sistema RSA é explorado pelo Ataque de Hastad's Broadcasts. Ele consiste em uma situação em que um emissor quer enviar a mesma mensagem para vários receptores distintos, cada um com sua respectiva chave pública. Descreveremos matematicamente esse caso e veremos que, caso um intruso intercepte a ligação, o ataque pode ser feito baseando-se no Teorema de Coppersmith, que apresentaremos à seguir.

3.4.1 Justificativa Matemática

Suponha-se que um emissor quer enviar uma mensagem encriptada M para os receptores $R_1, R_2, ..., R_k$. Cada um dos receptores tem a sua chave pública N_i, e_i. Assuma que $M < N_i \forall i$. O emissor encripta a mensagem M com as respectivas chaves públicas e envia o i-ésimo criptograma ao receptor R_i.

Primeiramente, para simplificar, suponhamos que todos os expoentes de encriptação $e = 3$. Pode-se mostrar que $k > 3$ permite que o intruso consiga recuperar M, a partir dos criptogramas $C_1, C_2, C_3, ...$, onde

$$C_1 \equiv m^3 \bmod N_1$$
$$C_2 \equiv m^3 \bmod N_2$$
$$C_3 \equiv m^3 \bmod N_3$$

Assuma que mdc $(N_i, N_j) = 1 \forall i, j$. (Caso contrário se poderia fatorar algum N_i). Aplicando o Teorema Chinês dos Restos[2] a C_1, C_2 e C_3 obtém-se $C \in Z_{N_1 N_2 N_3}$ que satisfaz

$$C \equiv m^3 \bmod N_1 N_2 N_3$$

Como $M < N_i \ \forall i \implies M^3 < N_1 N_2 N_3$. Então $0 < C, M^3 < N_1 N_2 N_3 \implies C = M^3$. Portanto, $M = \sqrt[3]{C}$

Concluímos, assim, que se os expoentes de encriptação forem todos iguais a e, pode-se recuperar M logo que se tenha $k > e$ onde k é o número de criptogramas interceptados.

[2] https://pt.wikipedia.org/wiki/Teorema_chinês_do_resto

Dessa forma, sabe-se que esse ataque só será bem sucedido se o expoente público e for relativamente pequeno.

Para prevenir ataques como esse, poderíamos tentar, além de utilizar expoentes e distintos, adicionar uma camuflagem (paddding) à mensagem antes de encriptografá-la.

Poderia-se camuflar da seguinte forma: se a mensagem M tem m-bits, aplicar a função linear $f_i(x) = i \cdot 2^m + M$ à mensagem M, ou seja, justapor os "bits"de i à esquerda dos de M, e encriptar $M_t = E_i(M)$.

Com isso, o emissor fixaria um polinômio público f_t em $Z_N[x]$ para cada receptor $R_1, \ldots, R_k$ e transmitiria a encriptação de $f_i(M)$ ao receptor R_j. O que seria interceptado seria então os criptogramas

$$d = f_i(M) \cdot e_i \mod N_i,$$

para $i = 1, 2, \ldots, k$.

O teorema à seguir mostra que mesmo usando expoentes de encriptação e distintos e a camuflagem por aplicação de funções polinomiais às mensagens, não se previne este tipo de ataque quando o número de criptogramas interceptados é suficientemente grande.

Teorema (Hastad) Sejam $N_1, N_2, \ldots, N_k$ inteiros dois a dois primos entre si, $N = N_1 N_2 \ldots N_k$ e $N_{\min} = \min_{i \neq j}\{N_i, N_j\}$. Sejam $f_1, f_2, \ldots, f_k$ polinômios de grau no máximo n, onde $n = \max_{i=1}^{k}\{\deg(f_i)\}$, e $P \in Z[x]$ é um polinômio público que serve para camuflar (pad) a mensagem. Suponhamos que existe um único $M < N_{\min}$ que satisfaz $f_i(M) \equiv 0 \pmod{N_i}$, para $i = 1, \ldots, k$. Se $k > n$, podemos recuperar M em tempo $O(n^4 \log^3 N)$.

Para demonstrar tal fato, é necessário abrir um parênteses para compreender o Teorema de Coppersmith:

Teorema de Coppersmith: Seja $f(x)$ um polinômio de grau k em uma variável,

$$f(x) = x^k + a_{k-1}x^{k-1} + \ldots + a_1 x + a_0.$$

Assumindo que $f(x)$ é um polinômio mônico irredutível, e N é um número composto difícil de fatorar. O teorema de Coppersmith fornece um método eficiente para, em certas condições, determinar todas as raízes pequenas de f modulo N.

Enunciando o Teorema: Sejam N um inteiro do qual não se conhece a fatorização e $f(x) \in Z[x]$ um polinômio mônico de grau k. Então, é possível encontrar todos os inteiros $|x_0| < N^{1/k}$ tais que $f(x_0) \equiv 0 \pmod{N}$ em tempo polinomial sobre $\log N$ e k.

Pode-se encontrar a demonstração nas referências.

Voltando à demonstração do Teorema de Hastad:

Prova: Primeiramente, observa-se que M é a mensagem a encriptar e, portanto, deverá ser de forma que $M < N$, ou seja, $M < \min_{i \neq j}\{N_i, N_j\}$. Notemos ainda que todos os polinômios f_t devem ter o coeficiente guia[3] invertível, pois caso contrário, isto é, se para algum i o coeficiente guia não for invertível em Z_N, ficaria exposta a fatoração de N. Dessa forma, todos os polinómios $g_t = f_t \bmod Z[x]$ têm também o coeficiente guia invertível.

[3]Coeficiente do termo de maior grau

Multiplicando cada g_t pela potência de x apropriada, podemos considerar que todos os polinómios têm grau n. Consideremos então

$$g_i(x) = \left(\prod_{a=1}^{n} a_{i,a} x^a \right) \pmod{N_i},$$

onde $a_{i,n}$ é invertível em Z_{N_i}. Aplicando o Teorema Chinês dos Restos, calculamos os números inteiros u_i $(i = 1, \ldots, k)$ tais que $u_i \equiv 1 \pmod{N_i}$ se $i = j$ e $u_i \equiv 0 \pmod{N_j}$ se $i \neq j$. Seja $G(x) \in Z[x]$ o polinômio definido por

$$G(x) = \sum_{i=1}^{k} u_i g_i(x).$$

Com isso,

$$G(M) \equiv \sum_{i=1}^{k} u_i g_i(M) \equiv g_i(M) \equiv 0 \pmod{N_i}, \quad \forall i,$$

o que implica que $G(M) \equiv 0 \pmod{N}$, porque $(N_i, N_j) = 1$, para todo $i \neq j$.

O coeficiente guia de $G(x)$ é $\prod_{i=1}^{k} a_{i,n} \pmod{N_i}$, para todo i, porque $\sum_{i=1}^{k} u_i a_{i,n} \equiv a_{i,n} \pmod{N_i}$. Como $(a_{i,n}, N_i) = 1$, para todo i, resulta que o coeficiente guia de $G(x)$ é invertível módulo N. Multiplicando $G(x)$ pelo inverso do coeficiente guia e reduzindo módulo N, obtemos um polinómio mónico $G(x)$ de grau n tal que $G(M) \equiv 0 \pmod{N}$ e, além disso, $M < N^{1-\frac{1}{n}} < N < N$. Finalmente, podemos aplicar o Teorema de Coppersmith e recuperar M, uma raiz inteira do polinômio mônico $G(M) \equiv 0 \pmod{N}$.

O tempo estimado para recuperar a mensagem M é determinado pelo tempo de aplicar o Teorema de Coppersmith que é $O(n^4 log^3 \bar{N})$

3.4.2 Implementação em Python

Como visto, não é necessário que os expoentes sejam iguais, e mesmo com a camuflagem de M, ainda é possível descriptografar. Mas, para ilustrar, será aqui implementado o primeiro caso citado, de quando os expoentes de encriptação são iguais a 3.

Funções usadas para o ataque:

```
1  import gmpy2
2  gmpy2.get_context().precision = 4096
3
4  from binascii import unhexlify
5  from functools import reduce
6  from gmpy2 import root, invert
7  from common_module_functions import PAndQFoundException
8  from utils import fatorar_em_primos
9  from math import gcd
10
11 EXPONENT = 3
12
```

```python
def chinese_remainder_theorem(items):
    # Determine N, the product of all n_i
    N = 1
    for a, n in items:
        N *= n

    # Find the solution (mod N)
    result = 0
    for a, n in items:
        m = N // n
        r, s, d = extended_gcd(n, m)

        # If the gcd is not 1, try to factorize n
        if d != 1:
            p = gcd(n, m)
            q = n // p
            raise PAndQFoundException(p, q)

        result += a * s * m

    # Make sure we return the canonical solution.
    return result % N


def extended_gcd(a, b):
    x, y = 0, 1
    lastx, lasty = 1, 0

    while b:
        a, (q, b) = b, divmod(a, b)
        x, lastx = lastx - q * x, x
        y, lasty = lasty - q * y, y

    return (lastx, lasty, a)


def mul_inv(a, b):
    b0 = b
    x0, x1 = 0, 1
    if b == 1:
        return 1
    while a > 1:
        q = a // b
        a, b = b, a % b
        x0, x1 = x1 - q * x0, x0
    if x1 < 0:
        x1 += b0
    return x1
```

```
61
62
63  def hastads_broadcast_attack(ciphertexts, modulus):
64      # modulus = [N1, N2, N3]
65      # ciphertexts = [C1, C2, C3]
66
67      C = chinese_remainder_theorem([(c, n) for c, n in zip(ciphertexts, modulus)])
68      M = int(root(C, 3))
69      return M
```

Abaixo está um exemplo da implementação funcionando. Uma mensagem foi criptografada 3 vezes com módulos diferentes mas mesmo expoente público=3. O resultado a seguir é o sucesso do ataque.

```
1   # Criptografando mensagem para teste
2
3   blocos_c1 = []
4   blocos_c2 = []
5   blocos_c3 = []
6
7   e = 3
8   n1 = 38509 # 97*397
9   n2 = 40067 # 103*389
10  n3 = 29993 # 89*337
11
12  # mensagem a ser criptografada
13  msg = ['f', 'u', 'i', '', 'd', 'e', 's', 'c', 'o', 'b', 'e', 'r', 't', 'o', '!', '!', '!']
14
15  for i in range(0, len(msg)):
16      msg[i] = str_to_int(msg[i])
17
18  for i in range(0, len(msg)):
19      c1 = (msg[i])**e%n1
20      c2 = (msg[i])**e%n2
21      c3 = (msg[i])**e%n3
22      blocos_c1.append(c1)
23      blocos_c2.append(c2)
24      blocos_c3.append(c3)
```

```
1   msg_descriptografada = []
2   p_e_q_encontrados = False
3   modulus = [n1, n2, n3]
4
5   for c1, c2, c3 in zip(blocos_c1, blocos_c2, blocos_c3):
6       ciphertexts = [c1, c2, c3]
```

```
    try:
        caractere = hastads_broadcast_attack(ciphertexts, modulus)
        caractere = chr(caractere)
        msg_descriptografada.append(caractere)
    except PAndQFoundException as err:
        print(f"Os fatores primos de N são p:{err.p} e q:{err.q}")
        p_e_q_encontrados = True
        p = err.p
        q = err.q
        break

if not p_e_q_encontrados:
    print("O ataque foi um sucesso!", end="\n")
    print("Mensagem descriptografada: ", end="")
    for i in msg_descriptografada:
        print(i, end="")

```

```
O ataque foi um sucesso!
Mensagem descriptografada: fui descoberto!!!
```

3.5 Ataque de Wiener

O ataque de Wiener é um método baseado em frações contínuas e é capaz de encontrar o expoente de decriptação quando o mesmo é pequeno.

Temos acesso a chave pública (e, N), e queremos descobrir o d tal que $ed \equiv 1 \pmod{\varphi(N)}$, primeiro notamos que:

$$\varphi(N) = (p-1)(q-1) = pq - (p+q) + 1$$

$$\varphi(N) \approx N$$

Afinal, como p e q são grandes o produto é muito maior com relação a $-(p+q)$ e o $+1$ também é negligenciável

Queremos $ed \equiv 1 \pmod{\varphi(N)}$, então:

$$ed = 1 + k \cdot \varphi(N)$$

$$ed - k \cdot \varphi(N) = 1$$

$$\frac{e}{\varphi(N)} - \frac{k}{d} = \frac{1}{d \cdot \varphi(N)}$$

$$\frac{e}{N} \approx \frac{k}{d}$$

Como conheço tanto e quanto N, se sou capaz de aproximar essa fração como outro número racional tenho a chance do d dessa aproximação ser o d usado pra descriptografar. Então

agora tenho o problema de dado e e N aproximar essa fração, para isso podemos usar a teoria de frações contínuas, em junção pra alguns critérios para tentar chegar nesse denominador.

Dada a fração $\frac{e}{N}$ é possível extrair a sequência $[a_0, a_1, a_2, ..., a_i]$ tal que:

$$\frac{e}{N} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \frac{1}{a_n}}}$$

truncando essas frações em um ponto nos retorna uma aproximação da fração inicial, por exemplo truncando ela até o a_3 obtemos:

$$\frac{e}{N} \approx a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \frac{1}{a_3}}}$$

e cada uma dessa aproximações pode ser manipulada pro formato de fração e temos uma possibilidade pro k e d que procuramos.

Na busca por essa aproximação podemos usar outros critérios pra julgar a aproximação em questão, podemos notar as seguintes coisas:

- Como $ed \equiv 1 \pmod{\varphi(N)}$, e $\varphi(N)$ é par, d deve ser ímpar, então ao olhar pras aproximações de $\frac{e}{N}$ não nos atentaremos aquelas em que d é par.
- Além disso $\varphi(N)$ é um inteiro, a aproximação $\frac{k}{d}$ que encontrarmos deve ser tal que $\frac{ed-1}{k} = \varphi(N)$ seja inteiro, afinal, caso contrário não tem como o d ser o expoente que estamos interessados.

Por último, servindo como critério definitivo, notamos que:

$$\varphi(N) = (p-1)(q-1) = pq - (p+q) + 1$$

$$\varphi(N) = N - (p+q) + 1$$

$$p + q = N - \varphi(N) + 1$$

Se olharmos pra equação quadrática $(x-p)(x-q) = 0$, que tem p e q como raízes, podemos desenvolver:

$$(x-p)(x-q) = 0$$

$$x^2 - (p+q)x + pq = 0$$

$$x^2 - (N - \varphi(N) + 1)x + N = 0$$

A partir do k e d que temos da aproximação, podemos calcular o $\varphi(N)$ substituir tudo na equação quadrática e caso as raízes que encontrarmos forem inteiras elas terão que ser p e q. Assim finalmente encontramos o d.

3.5.1 Exemplo:

Dada a chave píublica ($e = 42667, N = 64741$), é possível usar o Ataque de Wiener e encontrar d:

$$\frac{e}{D} = \frac{42667}{64741} = 0 + \frac{42667}{64741}$$

O a_0 nesse caso é 0, então a primeira aproximação pra fração é $\frac{0}{1}$, mas claramente $d = 1$ não servirá, continuando:

$$\frac{e}{D} = \frac{42667}{64741} = 0 + \frac{42667}{64741} = \frac{1}{\frac{64741}{42667}} = \frac{1}{1 + \frac{22074}{42667}} \approx \frac{1}{1}$$

$d = 1$, não servirá novamente, continuando:

$$\frac{e}{D} = \frac{1}{1 + \frac{1}{\frac{42667}{22074}}} = \frac{1}{1 + \frac{1}{1 + \frac{20593}{22074}}} \approx \frac{1}{1 + \frac{1}{1}} = 1/2$$

Como visto em um dos critérios, d não pode ser par, então prosseguimos pra próxima aproximação:

$$\frac{e}{D} = \frac{1}{1 + \frac{1}{1 + \frac{20593}{22074}}} = \frac{1}{1 + \frac{1}{1 + \frac{1}{\frac{22074}{20593}}}} = \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1481}{20593}}}} \approx \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}} = \frac{2}{3}$$

temos $k = 2$ e $d = 3$, podemos calcular $\varphi(N) = \frac{ed-1}{k} = \frac{42667 \cdot 3 - 1}{2} = 64000$, que deu inteiro, caso essa número não desse inteiro iríamos pra próxima aproximação, mas como foi inteiro checamos as raízes do polinômio contruído a partir de N e $\varphi(N)$:

$$x^2 - (N - \varphi(N) + 1)x + N = 0$$
$$x^2 - (64741 - 64000 + 1)x + 64741 = 0$$
$$x^2 - (64741 - 64000 + 1)x + 64741 = 0$$
$$x^2 - 742x + 64741 = 0$$

Pela fórmula quadrática as raízes desse polinômio são $x = 641$ ou $x = 101$, obtemos raízes inteiras, então basta que o seu produto seja o N para que tenhamos encontrado o d, e concluímos que sim, $641 \cdot 101 = 64741$, logo o $d = 3$

3.5.2 Implementação:

O algoritmo principal, sem a definição das funções auxiliares segue o mesmo princípio do exemplo. A função recebe como parâmetros os valores de N e e e retorna d.

```
1 def hack_RSA(e,n):
2     frac = rational_to_contfrac(e, n)
3     convergents = convergents_from_contfrac(frac)
4
```

```
    for (k,d) in convergents:

        #check if d is actually the key
        if k!=0 and (e*d-1)%k == 0:
            phi = (e*d-1)//k
            s = n - phi + 1
            # check if the equation x^2 - s*x + n = 0
            # has integer roots
            discr = s*s - 4*n
            if(discr>=0):
                t = is_perfect_square(discr)
                if t!=-1 and (s+t)%2==0:
                    #print("Hacked!")
                    return d
```

Exemplo de uso:

```
    print("Para N = 641*101 = 64741 e e = 42667:")
    print("d =",hack_RSA(42667,641*101))
```

Obtemos o resultado:

```
Para N = 641*101 = 64741 e e = 42667:
d = 3
```

Podemos usar essa função e achar todos os pares (e, d) para $N = 641 \cdot 101 = 64741$ que seriam encontrados por Wiener:

```
N = 641*101
phi_N = (641-1)*(101-1)
for i in range(1,phi_N):
    e = modInverse(i, phi_N)
    if e!= False:
        d = hack_RSA(e, N)
        if d != None:
            print("Pares encontrados: (d: ", d, ")", "(e: ", e, ")")
```

obtemos a saída:

```
Pares encontrados para N = : 64741
(d: 3 ) (e: 42667 )
(d: 7 ) (e: 9143 )
(d: 41 ) (e: 1561 )
(d: 35 ) (e: 1871 )
```

```
(d: 13 ) (e: 20149 )
(d: 5 ) (e: 13097 )
```

Testando com $N = 379 \cdot 751 = 284629$:

```
Pares encontrados para N = : 284629
(d:  11 ) (e:  103091 )
(d:  13 ) (e:  218077 )
(d:  17 ) (e:  33353 )
(d:  3 ) (e:  190507 )
(d:  33 ) (e:  43297 )
(d:  7 ) (e:  40823 )
(d:  37 ) (e:  30893 )
```

É possível mostrar que se $q < p < 2q$, e d obedece $d < \frac{1}{3}N^{\frac{1}{4}}$, dada a chave pública (e, N), o ataque de Wiener consegue encontrar d. Ou seja, é um ataque eficiente quando o expoente de descriptografia é pequeno.

3.6 Ataque cíclico

Pra esse ataque, após termos acesso a chave pública (e, N), escolhemos uma mensagem m qualquer e computamos m^e, m^{e^2}, m^{e^3}, m^{e^4}, ... até que a mensagem m que escolhemos apareça novamente, ou seja, encontramos k tal que $m^{e^k} = m$, e esse k vai funcionar pra qualquer mensagem que for mandada usando essa chave pública.

A primeira vista esse ataque parece uma forma mais esperta de tentar descriptografar uma mensagem, porém, por baixo dos panos ele acaba sendo só mais um algoritmo de fatorização de N, fomos bem vagos na nossa escolha tanto da mensagem m quanto do e, se assumirmos que é possível escolher m e e que tornam esse processo eficiente, é possível fatorar N também de forma eficiente.

Existem poucos valores de e cujo tamanho do ciclo módulo $\varphi(N)$ é curto. Em média o maior fator primo de $(p-1)$ terá o tamanho de cerca de 30% do tamanho de $p-1$, tentar usar esse ataque é tão dificil quanto fatorar o nosso N.

4 Como computadores quânticos quebram o RSA

Todos os ataques discutidos até então necessitavam de algum tipo de elo fraco na utilização do RSA para terem sucesso, seja uma escolha pobre dos expoentes, ou o uso de chaves públicas de forma irresponsável, que dão algum tipo de brexa, porém quando nennhuma dessas coisas acontecem sempre esbarramos no problema primoridal de fatorar um número de maneira eficiente, e o melhor algoritmo clássico que temos, o General number field sieve, não é capaz. Ainda não foi provado que é impossível achar um jeito mais eficiente de fatorar esses números com a computação clássica, porém é certo que alguém com um computador quântico funcional e com memória suficiente é uma ameaça a essa sistema graças ao Algoritmo de Shor.

4.1 Algoritmo de Shor

Temos como objetivo final descobrir algum dos fatores de N, podemos expandir um pouco essa tarefa estando abertos não só a fatores, mas a algum outro número que compartilha algum fator com N já que aí poderíamos usar o algoritmo de euclides encontrar esse fator, a idéia do algoritmo se baseia em dois passos:

Primeiro começamos com um palpite g para fator de N, checamos se $\text{mdc}(g, N) \neq 1$, provavelmente não será. A partir de g e N encontramos p de modo que $g^p \pm 1$ tem bem mais chances de ter algum fator em comum com N.

Isso está baseado no fato de que, se $\text{mdc}(g, N) = 1$, existe p tal que

$$g^p \equiv 1 \pmod{N}$$

daí:

$$g^p - 1 \equiv 0 \pmod{N}$$
$$(g^{p/2} + 1)(g^{p/2} - 1) \equiv 0 \pmod{N}$$
$$(g^{p/2} + 1)(g^{p/2} - 1) = m \cdot N$$

Nessa última expressão temos uma igualdade que se parece com uma fatoração de N e podemos usar euclides para encontrar os fatores em comum entre N e esse números, mas ainda temos 3 grandes problemas

- $(g^{p/2} + 1)$ ou $(g^{p/2} - 1)$ é múltiplo de N, se isso acontecer não seremos capazes de encontrar nenhum fator de N.
- A potência p que encontramos é ímpar, nesse caso nem $g^{p/2}+1)$ nem $(g^{p/2}-1$ seriam inteiros e não nos ajudariam na fatoriazação

Pra esses dois problemas, é possível notar que se começarmos com um g aleatório, espera-se que em 3/8 das tentativas nenhum desses ocorra, o que nos diz que com 10 tentativas temos 99% de chances de encontrar um p adequado. Porém o terceiro problema é o que realmente preucupa, como fazemos pra encontrar o expoente p, para um computador clássico e um problema tão ou mais difícil quanto fatorar o N, porém em um computador quântico podemos nos aproveitar de efeitos como a superposição e a interferência para conseguir esse p de forma eficiente.

4.2 Como encontrar p

O primeiro passo é começar com valor $|x\rangle$, e elevar nosso palpite inicial(g), a esse x, ficando com $|x, g^x\rangle$, vamos guardar tbm o valor de x, após isso olhamos pro resto desse g^x módulo N, temos a seguinte estrutura:

$$|x\rangle \rightarrow |x, g^x\rangle \rightarrow |x, g^x(modN)\rangle$$

em um computador clássico poderiámos fazer todas essas operações para um valor determinado de x, já aqui podemos começar com uma superposição de todos os possíveis expoentes:

$$|1\rangle + |2\rangle + |3\rangle + ... \rightarrow |1, g^1\rangle + |2, g^2\rangle + |3, g^3\rangle + ... \rightarrow |1, g^1(modN)\rangle + |2, g^2(modN)\rangle + ...$$

A partir disso queremos ser capazer de determinar o p, A partir do momento que observamos nossa saída ela irá colapsar em alguma das operações feitas, não temos sequer uma garantia de que o valor observado será o $|p, 1\rangle$, precisamos de algum jeito manipular isso de modo a sermos capazes de determinar o p, pra issno notamos que, se conhecêssemos o p:

$$g^p \equiv 0 \pmod{N}$$
$$\text{se } g^x \equiv r \pmod{N}$$
$$\text{então } g^{x+p} \equiv r \pmod{N}$$

e, da mesma forma

$$g^{x+k\cdot p} \equiv r \pmod{N} \ \forall k$$

A partir disso podemos usar essa propriedade e juntamente com propriedades da superposição e a partir de $|1, g^1(modN)\rangle + |2, g^2(modN)\rangle + |3, g^3(modN)\rangle + ...$ conseguir uma superposição onde, por exemplo, todos os restos são iguais a r:

$|a_1, r\rangle + |a_2, r\rangle + |a_3, r\rangle + ...$ e pela propriedade, a distância entre as potências é sempre igual a p, conseguimos usar g e N pra gerar uma superposição de potências com um período p entre elas, finalmente, nessa superposição é possível usar a transformada de fourier quântica e conseguir calcular o p.

Uma vez que temos o p checamos se ele é par e olhamos pra $g^{p/2} \pm 1$ em busca de fatores com o N.

4.3 Exemplo

Como o algoritmo de Shor fatora 314191:

O processo se inicia com um palpite g = 127. Começamos também com uma superposiçao de todos os possíveis expoentes:

$$|1\rangle + |2\rangle + |3\rangle + ...$$

elevamos 127 a cada um desses expoentes a partir da superposição:

$$|1, 127^1\rangle + |2, 127^2\rangle + |3, 127^3\rangle + ...$$

Calcula-se os restos módulo N:

$$|1, 127\rangle + |2, 16129\rangle + |3, 163237\rangle + ...$$

Medimos o resto dessa superposição e o sistema colapsa pra uma outra superposiçao onde todos os restos são iguais ao observado, por exemplo 74126:

$$|20, 74126\rangle + |4367, 74126\rangle + |8714, 74126\rangle + ...$$

Num cálculo real não haveria como saber quais numeros seriam esses, já que o sistema estaria sempre em uperposição, mas saberíamos que o período entre eles seria exatamente igual a p, nesse ponto aplicaríamos a transformada de Fourier quântica que nos tornaria capazes de descobrir qual é o p, nesse exemplo encontraríamos p = 17388 e assim podemos olhar para

$$127^{p/2} \pm 1 = 127^{17388/2} \pm 1 = 127^{8694} \pm 1$$

E finalmente podemos usar o algorítmo de euclides em N e $127^{8694} \pm 1$ e encontrar os fatores N = 314191 = 829·379

5 Resultados

Para testar e comparar os algoritmos apresentados, assim como verificar se as implementações foram feitas corretamente, foi feito uma parceria com outro projeto que implementa o algoritmo RSA para enviar mensagens criptografadas por e-mail. Diferentes situações foram simuladas para testar os diferentes tipos de ataque, conforme pode ser visto a seguir.

5.1 Pré-processamento dos dados

Antes de testar os algoritmos nos e-mails, foi necessário fazer uma preparação desses dados para que eles pudem ser acessados de maneira conveniente. Os emails enviados foram baixados e anexados ao repositório como extensão ".eml". A criptografia utilizada neles consiste em traduzir cada caractere de uma mensagem pela tabela ASCII, criptografar com o RSA cada um desses blocos e enviá-los separados por "_". A chave pública utilizada também foi enviada e pode ser acessada pois estava separada por "( )". Veja um exemplo de como os e-mails chegaram abaixo:

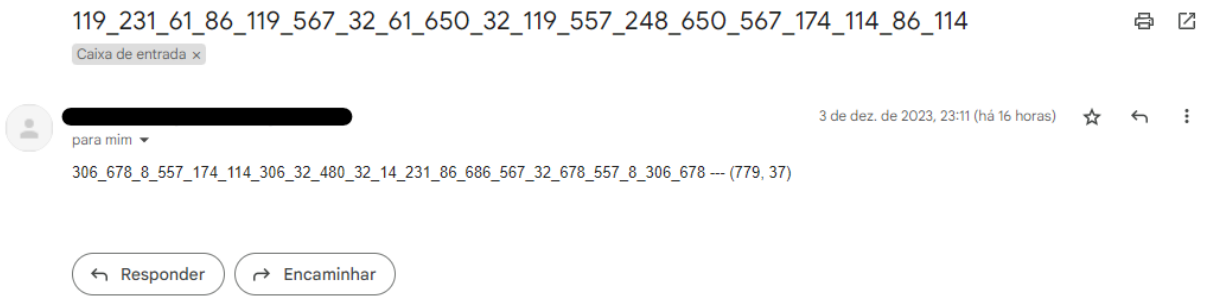


Figura 1: Formato do email recebido

Dessa forma, foi implantada uma função que lê o arquivo ".eml"e retorna um dicionário cujas chaves são: o módulo e o expoente público usados na criptografia, e duas listas distintas contendo o conjuto de blocos do corpo e do assunto do texto.

Com isso, foi possível implementar facilmente cada um dos algoritmos.

5.2 Ataque de Módulo Comum

O primeiro ataque a ser testado será o de Módulo Comum. Em uma empresa hipotética, imagine que o sistema de criptografia RSA foi implementado para mandar emails confidenciais aos funcionários. Numa tentativa de simplicar a administração das chaves desses funcionários, como vimos antes, pode ser que a mesma chave pública N seja distribuída a eles. Simularemos, então, o que aconteceria se um indivíduo tivesse acesso a dois emails iguais, com chaves privadas diferentes mas mesmo módulo N.

5.2.1 p e q pequenos

Nessa primeria situação, foram utilizados primos p e q pequenos para gerar N. Dessa forma, ao tentar atacar a mensagem com módulo comum, não foi possível inverter todos os blocos pois alguns deles tinham fatores em comum com N, justamente por conta desses fatores serem pequenos. Mas, uma vez encontrados fatores comuns com N, p e q foram

computados e a mensagem pode ser descriptografada a partir de $\phi(N) = (p-1)*(q-1)$. Observe abaixo:

```
from common_module_functions import PAndQFoundException

p_e_q_encontrados = False

n = email_common_modulus.get('email1.eml').get('N')
e1 = email_common_modulus.get('email1.eml').get('e')
e2 = email_common_modulus.get('email2.eml').get('e')
corpo_c1 = email_common_modulus.get('email1.eml').get('corpo')
corpo_c2 = email_common_modulus.get('email2.eml').get('corpo')

corpo_descriptografado = []
assunto_descriptografado = []

# Realiza o ataque no corpo dos emails
try:
    for c1, c2 in zip(corpo_c1, corpo_c2):
        caractere = attack(c1, c2, e1, e2, n)
        caractere = int_to_str(caractere)
        corpo_descriptografado.append(caractere)
except PAndQFoundException as err:
    print(f"Os fatores primos de N são p={err.p} e q={err.q}")
    p_e_q_encontrados = True
    p=err.p
    q=err.q

# Realiza o ataque no assunto dos emails
assunto_c1 = email_common_modulus.get('email1.eml').get('assunto')
assunto_c2 = email_common_modulus.get('email2.eml').get('assunto')

if not p_e_q_encontrados:
    try:
        for c1, c2 in zip(assunto_c1, assunto_c2):
            caractere = attack(c1, c2, e1, e2, n)
            caractere = int_to_str(caractere)
            assunto_descriptografado.append(caractere)

    except PAndQFoundException as err:
            print(f"Os fatores primos de N são p={err.p} e q={err.q}")
            p_e_q_encontrados = True
            p=err.p
            q=err.q
```

```
Não foi possível encontrar o inverso modular de 114 modulo 779, logo,
eles possuem um fator em comum.
```

```
Os fatores primos de N são p=19 e q=41
```

```
# Imprime a mensagem descriptografada
if p_e_q_encontrados:
    corpo_descriptografado = []
    assunto_descriptografado = []

    for c1 in corpo_c1:
        m = decrypt_pq(c1, p, q, e1)
        m = chr(m)
        corpo_descriptografado.append(m)

    for c1 in assunto_c1:
        m = decrypt_pq(c1, p, q, e1)
        m = chr(m)
        assunto_descriptografado.append(m)

print("Email Descriptografado!")
print("Assunto: ", end='')
for i in assunto_descriptografado:
    print(i, end='')
print('\nCorpo: ', end='')
for i in corpo_descriptografado:
    print(i, end='')
```

```
Email Descriptografado!
Assunto: duvido vc descobrir
Corpo: algebra é muito legal
```

5.2.2 p e q grandes

Como o ataque por módulo comum não foi necessário no caso acima, forams solicitados novos e-mails, mas, dessa vez, com p e q maiores. Verifique os resultados que seguem.

```

# Realiza o ataque no corpo dos emails
try:
    for c1, c2 in zip(corpo_c1, corpo_c2):
        caractere = attack(c1, c2, e1, e2, n)
        caractere = chr(caractere)
        corpo_descriptografado.append(caractere)
except PAndQFoundException as err:
    print(f"Os fatores primos de N são p={err.p} e q={err.q}")
```

```
    p_e_q_encontrados = True
    p=err.p
    q=err.q

# Imprime a mensagem descriptografada
if not p_e_q_encontrados:
    print("Email Descriptografado!")
    print("Corpo: ", end='')
    for i in corpo_descriptografado:
        print(i, end='')
else:
    corpo_descriptografado = []

    for c1 in corpo_c1:
        m = decrypt_pq(c1, p, q, e1)
        m = chr(m)
        corpo_descriptografado.append(m)

    print("Email Descriptografado!")
    print("Corpo: ", end='')
    for i in corpo_descriptografado:
        print(i, end='')
```

```
Email Descriptografado!
Corpo: duvido você descobrir
```

Pode-se notar que em ambos os casos foi possível descriptografar as mensagens. Isso indica que os algoritmos estão funcionando corretamente, mas também indica que utilizar módulos iguais para receptores diferentes não é seguro. Por isso, esse caso e é caracterizado como mal uso do sistema RSA.

5.3 Ataque de Hastad

Ainda no contexto da empresa que implantou o sistema RSA para enviar e-mails aos colaboradores: certamente, haverá casos em que a mesma mensagem é enviada para mais de um receptor. Isso justifica a tentativa de um ataque de Hastad, que é justamente quando vários receptores distintos recebem a mesma mensagem criptografada. Vimos anteriormente, com a prova do teorema, que mesmo em casos em que o expoente público e é distinto e a mensagem M é camuflada, ainda é possível descriptografar. Aqui, simularemos a situação em que os expoentes são iguais, apenas para ilustrar o método.

5.3.1 Um par de N's não coprimos

Nessa situação, os três e-mails recebidos tinham mesmo expoente público $e = 3$ e módulos N_1, N_2, N_3 distintos. Entretanto, ocorreu de que $mdc(N_i, N_j) \neq 1$ para algum par (i, j), e, dessa forma, não há como recorrer ao Teorema Chinês do Resto. Entretanto, com isso,

tivemos acesso a um dos fatores de algum N, e então puderam ser computados p e q e, outra vez, a mensagem pôde ser descriptografada a partir de $\phi(N) = (p-1) * (q-1)$. Veja os códigos abaixo.

```python
# Descripitografando a mensagem dos emails

n1 = email_hastads.get('email_pessoa1.eml').get('N')
n2 = email_hastads.get('email_pessoa2.eml').get('N')
n3 = email_hastads.get('email_pessoa3.eml').get('N')

corpo_c1 = email_hastads.get('email_pessoa1.eml').get('corpo')
corpo_c2 = email_hastads.get('email_pessoa2.eml').get('corpo')
corpo_c3 = email_hastads.get('email_pessoa3.eml').get('corpo')

corpo_descriptografado = []
assunto_descriptografado = []
p_e_q_encontrados = False
modulus = [n1, n2, n3]

for c1, c2, c3 in zip(corpo_c1, corpo_c2, corpo_c3):
    ciphertexts = [c1, c2, c3]
    try:
        caractere = hastads_broadcast_attack(ciphertexts, modulus)
        caractere = chr(caractere)
        corpo_descriptografado.append(caractere)
    except PAndQFoundException as err:
        print(f"Os fatores primos de N são p:{err.p} e q:{err.q}")
        p_e_q_encontrados = True
        p = err.p
        q = err.q
        break

if not p_e_q_encontrados:
    for c1, c2, c3 in zip(assunto_c1, assunto_c2, assunto_c3):
        ciphertexts = [c1, c2, c3]
        try:
            caractere = hastads_broadcast_attack(ciphertexts, modulus)
            caractere = chr(caractere)
            assunto_descriptografado.append(caractere)
        except PAndQFoundException as err:
            print(f"Os fatores primos de N são p:{err.p} e q:{err.q}")
            p_e_q_encontrados = True
            p = err.p
            q = err.q
            break
```

```
Os fatores primos de N são p:29 e q:251
```

```
e = 3

if p_e_q_encontrados:
    corpo_descriptografado = []
    assunto_descriptografado = []

    # descobrindo qual dos n's é o produto de p e q para saber qual
    # corpo e assunto descriptografar
    if p*q == n1:
        corpo_c = corpo_c1
        assunto_c = email_hastads.get('email_pessoa1.eml').get('assunto')
    elif p*q == n2:
        corpo_c = corpo_c2
        assunto_c = email_hastads.get('email_pessoa2.eml').get('assunto')
    elif p*q == n3:
        corpo_c = corpo_c3
        assunto_c = email_hastads.get('email_pessoa3.eml').get('assunto')

    for c in corpo_c:
        m = decrypt_pq(c, p, q, e)
        m = chr(m)
        corpo_descriptografado.append(m)

    for c in assunto_c:
        m = decrypt_pq(c, p, q, e)
        m = chr(m)
        assunto_descriptografado.append(m)

print("Email Descriptografado!")
print("Assunto: ", end='')
for i in assunto_descriptografado:
    print(i, end='')
print('\nCorpo: ', end='')
for i in corpo_descriptografado:
    print(i, end='')
```

```
Email Descriptografado!
Assunto: agora esse é impossível
Corpo: pois não é que vc descobriu mesmo
```

5.3.2 $N_i's$ coprimos

Por conta do ataque de Hastad não ter sido necessário no caso acima, novos e-mails foram solicitados dessa vez com a especificação de que os N_i precisavam ser coprimos. Veja os resultados.

```
corpo_descriptografado = []
p_e_q_encontrados = False
modulus = [n1, n2, n3]

for c1, c2, c3 in zip(corpo_c1, corpo_c2, corpo_c3):
    ciphertexts = [c1, c2, c3]
    try:
        caractere = hastads_broadcast_attack(ciphertexts, modulus)
        caractere = chr(caractere)
        corpo_descriptografado.append(caractere)
    except PAndQFoundException as err:
        print(f"Os fatores primos de N são p:{err.p} e q:{err.q}")
        p_e_q_encontrados = True
        p = err.p
        q = err.q
        break
```

```
if not p_e_q_encontrados:
    print("Email Descriptografado!")
    print("Corpo: ", end='')
    for i in corpo_descriptografado:
        print(i, end='')
```

```
Email Descriptografado!
Corpo: agora esse é impossível
```

É nótório mais uma vez que em ambos os casos pôde se descriptografar a mensagem, indicando o sucesso do ataque. As soluções que foram apresentadas para esse problema envolvem camuflar a mensagem e utilizar $e's$ distintos, mas foi visto que, mesmo assim, isso não previne esse tipo de ataque quando o número de criptogramas interceptados é suficientemente grande. Por isso, o RSA tem um ponto fraco quando se trata de enviar a mesma mensagem para muitos receptores diferentes.

5.4 Ataques baseados na fatoração

Até então, os ataques que testamos só são possíveis provenientes do mal uso do RSA. ....

Dentre os exemplos fornecidos tivemos chaves que era relativamente pequenas como:

- (779, 37)
- (681, 3)
- (7279, 3)

Pra todas essas os métodos que envolviam fatorar o N funcionaram e permitiam descriptografar as mensagens, além dessas também foi enviada uma mensagem que usava um N com cerca de 1200 dígitos, pra esse N nenhum dos ataues envolvendo fatoração foi capaz de decifrar, já que além da enormidade do número o p e o q não foram escolhidos de forma insegura.

6 Conclusão

Em relação à segurança do RSA, é crucial destacar que os ataques bem-sucedidos muitas vezes resultam de práticas descuidadas na utilização do algoritmo. Quando as pessoas negligenciam medidas de segurança ao enviar e receber mensagens de várias fontes ou fazem escolhas inadequadas para os parâmetros como p, q e d, os sistemas podem se tornar vulneráveis a ameaças.

No entanto, quando todos os cuidados são tomados de maneira apropriada, esses ataques tendem a se equiparar à complexidade de fatorar números. É importante mencionar que, até o momento, nosso melhor método para fatorar números é bastante limitado. O algoritmo de Shor como vimos pode fatorar de forma eficiente, porém a barreira tencológica imposta sobre o algoritmo de Shor e sobre outros algoritmos ainda mais eficientes que fazem uso de recursos quânticos mostram que para essas técnicas realmente se tornarem uma ameaça séria ao RSA seria necessário um avanço tecnológico muito grande, e além disso já existem discussões sobre a criptografia pós quântica que não se baseia no problema da fatoração para ser segura e sim em outras técnicas.