FUNDAÇÃO GETULIO VARGAS
ESCOLA DE MATEMÁTICA APLICADA
Álgebra & Criptografia

Ataque ao RSA

ERIC MANOEL RIBEIRO DE SOUSA
LUAN RODRIGUES DE CARVALHO
RODRIGO SEVERO ARAÚJO
VICTOR GABRIEL HARUO IWAMOTO

Rio de Janeiro – RJ
Dezembro 2025

1 Introdução

O presente trabalho consolida os fundamentos teóricos da disciplina de Álgebra e Criptografia através de uma análise prática da segurança do sistema RSA. Embora matematicamente robusto, o RSA torna-se vulnerável quando parâmetros são escolhidos de forma inadequada. Nesse contexto, este estudo examina a eficiência e a complexidade de implementação de diferentes vetores de ataque explorando falhas estruturais e de configuração. A análise comparativa abrange desde métodos genéricos, como a Força Bruta e a Fatoração de Fermat, até técnicas específicas baseadas em propriedades algébricas, incluindo o Ataque de Módulo Comum, os Algoritmos de Pollard, o Ataque ao Pequeno Expoente Público e o Ataque de Wiener.

2 A criptografia RSA

A criptografia RSA é um sistema de encriptação que segue o sistema de chave pública e privada. Em resumo, o RSA envolve um par de chaves, uma *chave pública* que pode ser conhecida por todos e uma *chave privada* que deve ser mantida em sigilo. Toda mensagem cifrada só pode ser decifrada usando a respectiva chave privada.

O funcionamento da criptografia RSA é simples:

2.1 Encriptação

Queremos codificar uma mensagem $m \in \{0, 1, 2, \ldots, n-1\}$

1. Escolha p e q primos grandes.
2. Obtenha $n = p \cdot q$
3. Escolhemos e tal que $1 < e < \phi(n)$ e $\text{mdc}(e, \phi(n)) = 1$
4. $c \equiv m^e \pmod{n}$ (c é a mensagem criptografada)

A sua chave pública é o par (n, e)

2.2 Decriptação

1. Precisamos de uma chave privada d tal que $e \cdot d \equiv 1 \pmod{\phi(n)}$
2. Pois $c^d \equiv (m^e)^d = m^{ed} = m^{k\phi(n)+1} = (m^{\phi(n)})^k \cdot m = m \pmod{n}$

Podemos fazer um exemplo numérico para termos uma ideia de como o sistema funciona (com números pequenos para facilitar os cálculos).

Exemplo. Dada uma mensagem m tal que $0 \leq m < 1219$ e a mensagem criptografada é $c = m^{35} \pmod{1219}$. Qual o valor do expoente descriptografador d?

Primeiramente perceba que $n = 1219 = 23 \cdot 53$, logo, $\phi(n) = 22 \cdot 52 = 1144$. Portanto, basta encontrar d tal que $35 \cdot d \equiv 1 \pmod{1144}$, ou seja, basta resolvermos a equação diofantina

$$1144x + 35d = 1$$

Daí, via algoritmo de Euclides estendido, obtemos que $d = 523$ é o expoente descriptografador.

O exemplo acima mostra que, para "quebrarmos"o RSA, basta encontrar os primos p e q que compõem n. Boa parte dos ataques a seguir se basearão nessa ideia.

3 Fatoração (Naive Algorithm)

Uma abordagem natural contra o sistema criptogáfico RSA consiste na fatoração da chave pública n (um semiprimo) com fim de encontrar os fatores primos p e q. Por conseguinte, descobre-se a classe de congruência $\phi(n) = (p-1)(q-1)$ que daí basta determinar d - inverso multiplicativo de e - resultando na completa quebra da criptografia e, portanto, a exposição da mensagem.

Apesar disso, a fatoração de números inteiros é um problema computacionalmente custoso visto que hodiernamente o sistema RSA adota como padrão mínimo 1024 bits como o tamanho da chave pública, equivalente a cerca de 300 dígitos. Ademais, os algoritmos tendem a ter maior dificuldade para encontrar números semiprimos como no caso do RSA, daí com as tecnologias disponíveis em estimativa demora de 10 à 15 anos de computação para descriptografar com tal método.

Logo, a fatoração de números inteiras é virtualmente inofensivo e inviável para um ataque sério ao RSA, entretanto, é útil no âmbito acadêmico dado que é uma ótima base de comparação com os demais ataques.

3.1 Implementação Computacional

O algoritmo de "força bruta"para fatoração é conhecido como Divisão por Tentativa (Trial Division). Embora sua premissa seja simples, é possível otimizá-lo para contornar casos desnecessários como:

1. **Limite de $\sqrt{n}$:** Basta testar divisores até a raiz quadrada de n. Se n possuir um fator $a > \sqrt{n}$, ele obrigatoriamente terá um fator $b < \sqrt{n}$ (tal que $n = ab$), que já teria sido encontrado.

2. **Tratamento do fator 2:** O número 2 é o único primo par. Ele pode ser tratado em um loop separado, o que permite que o loop principal teste apenas divisores ímpares.

3. **Teste de ímpares:** Após remover todos os fatores 2, o n restante é ímpar. Seus fatores primos também serão ímpares. Portanto, o loop principal pode testar apenas divisores a partir de 3, incrementando o divisor de 2 em 2 (3, 5, 7, ...).

```python
def naive(n):
    begin = time.time()
    div = []
    while (n%2 == 0):
        div.append(2)
        n = n//2
    i = 3
    while i*i <= n:
        print(i)
        while n%i == 0:
            div.append(i)
            n = n//i
        i = i + 2

    end = time.time()
    if n > 1:
        div.append(n)
    return div, end-begin
```

Listing 1: Naive Algorithm

3.2 Simulação Computacional

Com o intuito de compreender a natureza do método, realizou-se testes computacionais focados no pior caso do algoritmo: a fatoração de semiprimos. O experimento consistiu em variar o número de dígitos do semiprimo e comensurar o tempo de execução decorrido.

Digitos	Tempo (s)
1	2.38×10^{-6}
6	5.69×10^{-4}
11	0.30
16	123.84

Tabela 1: Amostragem dos Tempos de Fatoração (a cada 5 itens)

Tabela 2: Semiprimos e seu respectivo número de dígitos

Semiprimo (n)	N° de Dígitos
6	1
77	2
989	3
2291	4
97627	5
358091	6
8846573	7
63451711	8
553789213	9
5276275391	10
48965927779	11
868082737663	12
5163693436199	13
53684551531801	14
635621477042171	15
6750421608780299	16
68569780649272979	17

Por meio dos dados presentes na tabela, é plausível julgar um crescimento exponencial. Essa natureza é visualmente confirmada nos gráficos da Figura 1.

O gráfico da esquerda (Figura 1) plota os dados em escala linear; a curva explode de tal forma que os primeiros pontos se tornam indistinguíveis, um comportamento clássico de crescimento exponencial.

O gráfico da direita (Figura 2), por sua vez, aplica uma escala logarítmica ao eixo Y (Tempo). Como esperado de uma função exponencial, os pontos se alinham em uma reta, confirmando a relação $Tempo \approx e^k$, onde k é o número de dígitos.

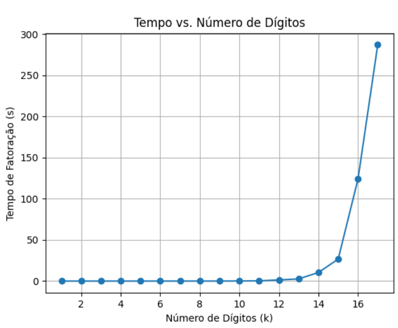

Figura 1: Sem Escala Logarítmica.

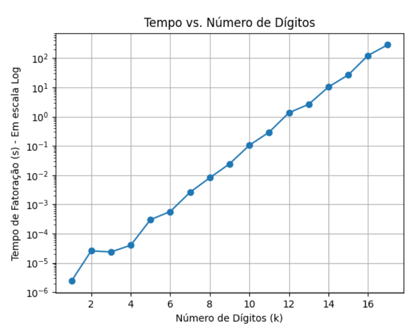

Figura 2: Escala Logarítmica.

O resultado final é categórico: o algoritmo levou 287 segundos (mais de 4 minutos e meio) para fatorar um semiprimo de apenas 17 dígitos. Considerando que chaves RSA (que são semiprimos) utilizavam como padrão mínimo 1024 bits (cerca de 300 dígitos), fica evidente que o método de divisão por tentativa é computacionalmente inviável para qualquer aplicação criptográfica real.

4 Fatoração de Fermat

Em particular, suponha que haja o conhecimento prévio de que a fatoração da chave pública sejam números próximos de $\sqrt{n}$, mostraremos um método eficiente desenvolvido pelo matemático francês Pierre de Fermat (1601-1665) capaz de descobrir p e q com voracidade.

4.1 Método

Seja n um número inteiro maior do que 1 que desejamos fatorar. Note que, o objetivo principal do método é encontrar inteiros não negativos x e y, tais que $n = x^2 - y^2$; já que é conhecida a identidade $x^2 - y^2 = (x - y)(x + y)$, daí encontramos uma fatoração de n que não necessariamente é em fatores primos. Entretanto, como no sistema RSA trabalhamos com semiprimos, o método encontra uma fatoração em primos.

4.1.1 Algoritmo de Fatoração

Seja n um inteiro ímpar maior do que 1.
Passo 1: Calcule $\sqrt{n}$

- Se $\sqrt{n}$ for um número inteiro, então o processo terminou, pois n é um quadrado perfeito e basta, então, tomarmos $x = \sqrt{n}$ e $y = 0$.
- Se $\sqrt{n}$ não for um número inteiro, defina $\lfloor\sqrt{n}\rfloor$ e siga para o **Passo 2**

Passo 2: Faça $x = b + 1$.

- Se $x = \frac{n+1}{2}$, então n é primo, $y = \sqrt{x^2 - n}$ e o processo terminou. Nesse caso, observe que

$$\begin{aligned} y &= \sqrt{x^2 - n} \\ &= \sqrt{\left(\frac{n+1}{2}\right)^2 - n} \\ &= \frac{n-1}{2}. \end{aligned}$$

- Se $x \neq \frac{n+1}{2}$, então siga para o **Passo 3**

Passo 3: Calcule $y = \sqrt{x^2 - n}$.

- Se y for um número inteiro, então o processo terminou pois n é composto e obtivemos inteiros não negativos x e y tais que $x > y$ e $n = x^2 - y^2$.

- Se y não for um número inteiro, desconsidere o valor anterior de b, defina $b = x$ e volte para o **Passo 2**, redefinindo x conforme este novo valor de b.

4.1.2 Por que o método funciona?

Para demonstrar o funcionamento do algoritmo, dividiremos a prova em passos, mostrando o funcionamento para números compostos que não são quadrados perfeitos e provando que o algoritmo é finito.

Na demonstração, consideramos que n é um número ímpar. Caso contrário, seria possível escrever $n = 2^a \cdot b$, com a inteiro positivo e b inteiro positivo ímpar. Dado que 2 é primo, bastaria fatorar b para descobrir a fatoração de n em números primos, recaindo assim no problema de fatorar números ímpares.

Dessa maneira, vamos mostrar que, para o caso em que n é inteiro composto, ímpar e maior do que 1, sempre encontraremos o valor de x.

Seja n um inteiro composto, ímpar, maior do que 1 e que não seja um quadrado perfeito. Seja $n = a \cdot b$ uma fatoração para n, com a e b inteiros tais que $1 < a < b < n$. Note que $a \neq b$, já que estamos supondo que n não é quadrado perfeito; além disso, a e b devem ser ímpares.

Devemos assegurar a existência de números inteiros positivos x e y tais que $n = x^2 - y^2$.

$$\begin{cases} n = a \cdot b \\ n = x^2 - y^2 \end{cases}$$

Assim, temos que $a \cdot b = (x - y)(x + y)$. Tomemos $a = x - y$ e $b = x + y$. Logo:

$$\begin{cases} x = \frac{a+b}{2} \\ y = \frac{b-a}{2} \end{cases}$$

Verifiquemos que os valores de x e y satisfazem todas as condições necessárias:

1. Como $1 < a < b < n$, então $a + b > 0$ e $b - a > 0$. Assim, x e y são positivos.

2. Como a e b são ímpares, a soma $a + b$ e a diferença $b - a$ são pares. Logo, x e y são inteiros.

3. Note que:

$$x^2 - y^2 = \left(\frac{a+b}{2}\right)^2 - \left(\frac{b-a}{2}\right)^2 = \frac{(a+b)^2 - (b-a)^2}{4} = \frac{4ab}{4} = a \cdot b = n$$

   Daí, $n = x^2 - y^2$, o que implica $y = \sqrt{x^2 - n}$, dado que y é positivo.

4. Sabemos que, para $a \neq b$, $(\sqrt{a} - \sqrt{b})^2 > 0$. Assim, $a - 2\sqrt{ab} + b > 0$, o que implica $\sqrt{ab} < \frac{a+b}{2}$. Dessa forma, $\sqrt{n} < x$. Portanto, $x > \lfloor\sqrt{n}\rfloor$. Como x é inteiro, temos que $\lfloor\sqrt{n}\rfloor + 1 \leq x$.

5. Sendo $b > a \geq 2$ (pois n é composto ímpar, logo os fatores são ≥ 3), temos:

   - $2 < b \implies a = \frac{a}{2} \cdot 2 < \frac{a}{2} \cdot b = \frac{ab}{2}$. Logo, $a < \frac{ab}{2}$.
   - De $2 \leq a$, segue que $b = 2 \cdot \frac{b}{2} \leq a \cdot \frac{b}{2} = \frac{ab}{2}$. Logo, $b \leq \frac{ab}{2}$.

   Somando as desigualdades:

$$a + b < \frac{ab}{2} + \frac{ab}{2} = ab < ab + 1$$

   Dividindo por 2:

$$\frac{a+b}{2} < \frac{ab+1}{2} = \frac{n+1}{2}$$

   Portanto, $x < \frac{n+1}{2}$.

6. Note que $x - y = \frac{a+b-(b-a)}{2} = \frac{2a}{2} = a$. Como n é composto, temos $a > 1$, logo $x - y > 1$.

Assim, o algoritmo funciona para n composto e não quadrado perfeito.

Sabemos que, se n é composto, encontraremos um valor adequado para x tal que $x < \frac{n+1}{2}$. Portanto, se x atingir o valor $\frac{n+1}{2}$ no decorrer do processo sem encontrar uma fatoração, significa que n não é composto. Como o algoritmo é aplicado para um inteiro ímpar maior do que 1, não sendo composto, n será primo. Dessa maneira, demonstramos também que o algoritmo é finito.

4.2 Implementação Computacional

```
def fermat(N):
    x = math.isqrt(N)

    if (N%2 == 0):
        return (2, N//2)

    if (x*x == N):
        return (x, x)

```

```python
    while x != (N + 1)//2:
        x = x + 1
        w = pow(x, 2) - N
        y = math.isqrt(w)
        if (y*y  == w):
            return (x-y, x+y)

    return (1, N)
```

Listing 2: Fermat Algorithm

4.2.1 Simulação

Como comentado no início da seção, a Fatoração de Fermat é um mecanismo útil quando os fatores do número são próximos, entretanto, vamos analisar o quão melhor o algoritmo se torna ao tomar o pior e melhor caso; isto é, no pior caso tomar um semiprimo $n = 3 \cdot p$ - tomamos 3 pois o algoritmo trata o 2 como um caso particular e por isso não segue diretamente o algoritmo - e o melhor caso em que p e q são os primos mais próximos. Além disso, incrementamos os digitos do número e vemos o tempo decorrido para calcular a decomposição em primos.

Tabela 3: Tabela para Fatoração de Semiprimos

Semiprimo (N)	Fatores (p, q)
15	3·5
143	11·13
2491	47·53
47053	211·223
304679	547·557
5494327	2341·2347
76562491	8747·8753
816359183	28571·28573
7844290183	88547·88589
83274953467	288571·288577
150986644891	388567·388573

Tabela 4: Tabela para Fatoração de Semiprimos

Semiprimo (N)	Fatores (p, q)
93	$3 \cdot 31$
267	$3 \cdot 89$
1569	$3 \cdot 523$
19473	$3 \cdot 6491$
499461	$3 \cdot 166487$
2899473	$3 \cdot 966491$
17899473	$3 \cdot 5966491$
107899431	$3 \cdot 35966477$
1607899443	$3 \cdot 535966481$
13607899449	$3 \cdot 4535966483$
874607899119	$3 \cdot 291535966373$

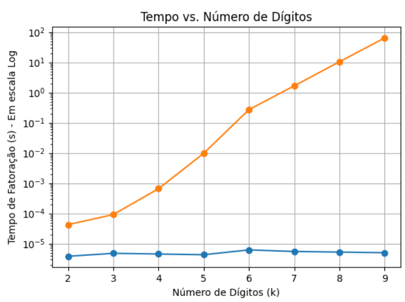

Figura 3: Sem Escala Logarítmica.

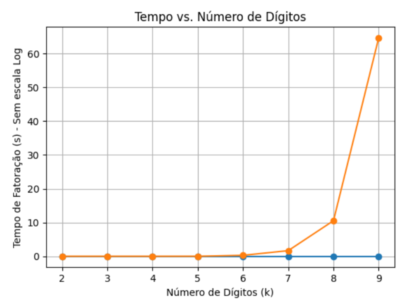

Figura 4: Escala Logarítmica.

Logo, para números primos próximos a fatoração de Fermat é indiscutivelmente boa, não obstante, a eficiência decaí exponencialmente quando os primos se distam.

4.3 Afinal, Qual é Melhor?

Pela simulação anterior é visível a eficiência da Fatoração de Fermat para números primos próximos, entretanto, vimos que para primos distantes é um algoritmo computacionalmente ruim, então é factível indagar se ainda assim consegue ser melhor do que utilizar a força bruta. Além disso, quão melhor é para números primos próximos?

Destarte, segue uma simulação comparando os tempos computacionais do Algoritmo de Fatoração de Fermat em relação ao Força Bruta.

Para Números Primos Próximos

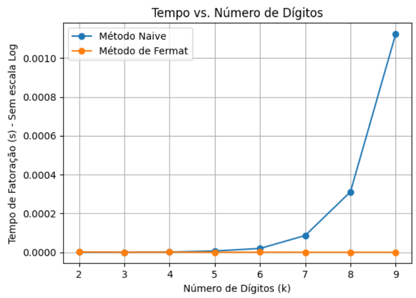

Figura 5: Sem Escala Logarítmica.

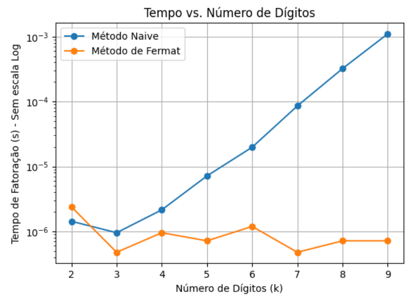

Figura 6: Escala Logarítmica.

Para Números Primos Distantes

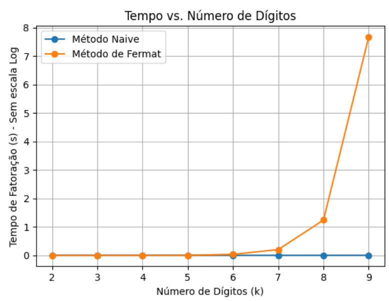

Figura 7: Sem Escala Logarítmica.

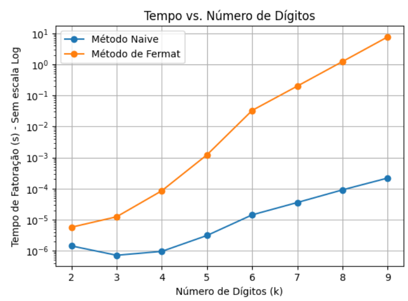

Figura 8: Escala Logarítmica.

Por análise visual, vemos que para valores primos próximos - como esperado - o algoritmo de Fermat é superior; em contra partida, para primos distantes apesar da força bruta ser um algoritmo ineficiente ainda assim, consegue superar o de Fermat. Em suma, o Algoritmo de Fatoração de Fermat não é um método perfeito, isto é, não podemos/devemos utilizá-lo em qualquer situação indiscriminadamente, é necessário que haja um conhecimento prévio de que o RSA tem uma falha estrutural ao ter números da chave pública com valores próximos.

5 Ataque Pollard $p-1$

Entramos novamente no problema de fatorar $n = p \cdot q$ onde p e q são primos grandes. Este método, em particular, é altamente eficiente quando $p-1$ tem fatores primos pequenos.

5.1 Método

Seja $n = pq$, com p, q primos. O Pequeno Teorema de Fermat garante que

$$a^{p-1} \equiv 1 \pmod{p}$$

para todo a que seja primo com p. Mas na prática, não sabemos o valor de p, veremos o que acontece em breve. Suponha que $p-1$ seja o fator de algum número L. Então $L = (p-1)k$, logo:

$$a^L \equiv (a^{p-1})^k \equiv 1 \pmod{p}$$

Consequentemente, p divide $a^L - 1$, e uma vez que p é um fator de n, segue que o mdc de $a^L - 1$ e n tem o fator p.

Problema: Como encontrar L?

Para fatorar algum número n, escolha a relativamente primo com n. Então:

- Calcule $a^{k!} \pmod{n}$ para $k = 1, 2, \ldots$ até algum limite (B).
- Encontre o mdc de $(a^{k!} - 1) \pmod{n}$ e n.
- Qualquer mdc não trivial é um fator de n.

Exemplo. Fatore 1403 usando o método $p-1$ de Pollard.

Tomando $a = 2$ e calculando $2^{k!} \pmod{1403}$ para $k = 2, 3, 4, \ldots$ e encontrando $mdc(2^{k!} - 1, 1403)$

$$\begin{aligned}
2^{2!} &\equiv 4 \pmod{1403} & mdc(4-1, 1403) &= 1 & &\Rightarrow \text{Continuamos} \\
2^{3!} &\equiv 64 \pmod{1403} & mdc(64-1, 1403) &= 1 & &\Rightarrow \text{Continuamos} \\
2^{4!} &\equiv 142 \pmod{1403} & mdc(142-1, 1403) &= 1 & &\Rightarrow \text{Continuamos} \\
2^{5!} &\equiv 794 \pmod{1403} & mdc(794-1, 1403) &= 61 & &\Rightarrow \textbf{Achamos!}
\end{aligned}$$

Daí, encontramos o fator $p = 61$, donde obtemos $1403 = 61 \times 23$.

5.2 Implementação Computacional

A Implementação computacional segue as mesmas linhas que a explicação teórica feita anteriormente. Apenas comentaremos alguns detalhes que não foram descritos até então. O código está disponível abaixo, e será detalhado a seguir.

```python
def pollard(n):
    # Testando várias bases, quando necessário
    bases = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]

    B = int(n**0.20) # limite para k (usaremos k!)

    for base in bases:
        a = base
        for k in range(2, B + 1):
            a = pow(a, k, n) # a^k (mod n)
            d = math.gcd(a - 1, n)
```

```python

            if 1 < d < n:
                return d, n // d

            elif d == n:
                print(f"  -> Falha com base {base} (MDC deu n). Trocando base...")
                break

    return None, None

p,q = pollard(z)

if p == None and q == None:
    print("Falhamos!")
```

Listing 3: Algoritmo Pollard $n-1$

Os detalhes são o que segue:

1. O código começa com uma lista de bases para o qual possivelmente podemos testar o algoritmo, essa lista serve para caso em algum momento da iteração o mdc entre n e $a^{k!}-1 \pmod n$ seja n, sinal que devemos mudar de base.

2. Definimos também um valor B que será o limite de iterações possíveis para k. Esse valor B irá cresce de acordo com o tamanho de n, usaremos $B = n^{0.20}$ para como heurística inicial. Para alguns números muito grandes, como por exemplo, $n = 68569780649272979$, se $B = n^{0.20}$, o algoritmo falha (pois $p-1$ não possui fatores primos pequenos), porém, se aumentarmos o valor do expoente para 0.25, o algoritmo já funciona perfeitamente.

3. A partir daí, iteramos e fazemos os cálculos como descrito anteriormente

5.2.1 Simulação

Agora iremos comparar nosso algoritmo pollard $p-1$ com o algoritmo de força bruta. Como queremos verificar a possível eficiencia do algoritmo, começaremos com 5 dígitos e iremos até 18. Além disso, foram escolhidos semiprimos de forma aleatória, e para números gerados aleatoriamente, a probabilidade de $p-1$ ter fatores primos pequenos é relativamente baixa. Por isso, tivemos que aumentar B drasticamente ($n^{0.47}$) para forçar o algoritmo a funcionar, o que atrapalhou um pouco sua performance comparada à teoria.

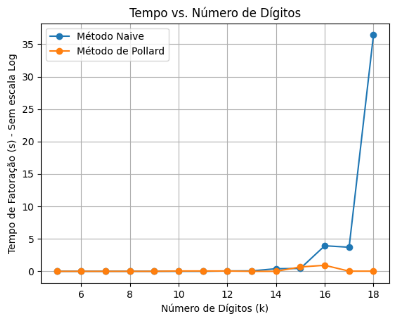

Figura 9: Sem Escala Logarítmica.

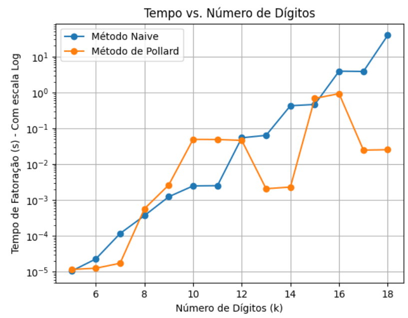

Figura 10: Escala Logarítmica.

Perceba que quando os dígitos ainda não são consideravelmente grandes, ainda ficamos na dúvida qual algoritmo é mais eficiente, a partir do 15^o já notamos uma maior vantagem para o método de pollard $p-1$. Além disso, como os semiprimos foram escolhidos de forma aleatória, não percebemos uma vantagem tão consistente e notória ao método de pollard.

Agora, escolhemos semiprimos $n = p \cdot q$ tais que $p-1$ possuem fatores primos suficientemente pequenos (chamaremos $p-1$ de $B-$suave, isto é, o maior fator primo de $p-1$ é menor que B), vejamos os resultados encontrados:

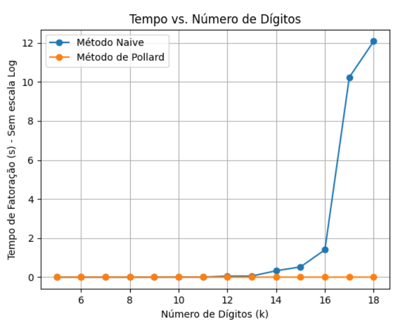

Figura 11: Sem Escala Logarítmica.

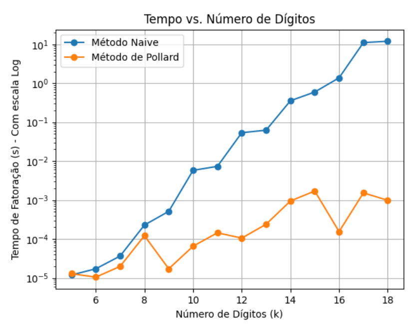

Figura 12: Escala Logarítmica.

Agora a vantagem do método de pollard é bem mais notória, ganhando praticamente em todos os números de dígitos.

5.3 Conclusão

Podemos perceber pelos resultados e pela discussão acima que o algoritmo de Pollard $p-1$ é um excelente algoritmo em comparação a força bruta para quando temos muitos dígitos em n. E mais ainda, quando o fator primo p de n é tal que $p-1$ é $B-$suave. Assim, apesar de um bom método, ele se mostra relativamente limitado. Porém sobre certas condições como a suavidade de $p-1$, ele é altamente eficiente.

6 Ataque do Módulo Comum

Sabemos que, na criptografia RSA, cada usuário torna público o par (e, n). Considere o caso em que dois usuários compartilham indevidamente o mesmo módulo n. Assim, suas chaves públicas são (e_1, n) e (e_2, n).

Suponha que uma mesma mensagem m seja enviada para ambos os usuários. Pela criptografia RSA, as cifras recebidas são

$$c_1 \equiv m^{e_1} \pmod n \qquad \text{e} \qquad c_2 \equiv m^{e_2} \pmod n.$$

Suponha agora que um atacante intercepte (c_1, c_2, e_1, e_2, n). Mostraremos que, se $\gcd(e_1, e_2) = 1$, então é possível recuperar a mensagem m *sem fatorar* n, independentemente de seu tamanho.

6.1 Recuperação da mensagem

Como $\gcd(e_1, e_2) = 1$, pela Identidade de Bézout existem inteiros a e b tais que

$$ae_1 + be_2 = 1.$$

Então,

$$m \equiv m^1 \equiv m^{ae_1+be_2} \equiv (m^{e_1})^a (m^{e_2})^b \pmod n$$

Caso algum dos coeficientes a ou b seja negativo, utiliza-se o inverso modular correspondente, que pode ser obtido pelo algoritmo estendido de Euclides. Portanto, o atacante pode então computar

$$m \equiv c_1^a \cdot c_2^b \pmod n,$$

sem necessidade de fatorar n.

6.2 Conclusão

Assim, se dois usuários compartilham o **mesmo módulo** n e cifram a mesma mensagem com expoentes públicos **coprimos** e_1 e e_2, então qualquer atacante que intercepte (c_1, c_2, e_1, e_2, n) pode recuperar a mensagem m, violando completamente a segurança do RSA nesse cenário.

7 Ataque da Pequena Chave Pública

Como em todo processo, sempre buscamos maior rapidez e eficiência, assim, podemos ficar bastante tentados a possuir um expoente público pequeno, para que, no processo de encriptação $c \equiv m^e \pmod n$, a demora computacional seja reduzida.

De fato, parece tentador e seguro, já que a *dificuldade* de quebra do RSA se deve à fatoração de n, então diminuir o expoente parece razoável.

Entretanto, existe uma limitação nesse processo, uma vez que, caso $m^e < n$, a criptografia $c \equiv m^e \pmod{n} \Rightarrow c = m^e$, então basta

$$m = \sqrt[e]{c}$$

para recuperar a mensagem e novamente **sem precisar fatorar n.**

7.1 O Ataque de Hastad

Suponhamos que um emissor deseja enviar uma mensagem encriptada M para os receptores $R_1, R_2, \cdots, R_k$. Cada um dos receptores tem a sua chave pública (N_i, e_i). Vamos assumir que M é menor que qualquer dos N_i. Um intruso pode interceptar a ligação sem que o emissor perceba e coletar os k criptogramas.

Simplificando, considere que todos possuem $e_i = 3$. Podemos mostrar que se $k \geq 3$, o intruso consegue recuperar a mensagem M a partir de $C_1, C_2, C_3, \cdots$ criptografia da mensagem em cada emissão.

Pois veja, se temos:

$$\begin{cases} C_1 \equiv m^3 \pmod{N_1} \\ C_2 \equiv m^3 \pmod{N_2} \\ C_3 \equiv m^3 \pmod{N_3} \end{cases}$$

Temos também que $\gcd(N_i, N_j) = 1 \quad \forall i, j$ (Ou seja, coprimos, pois caso não fossem, poderíamos encontrar a fatoração deles). E pelo Teorema do Resto Chinês (TRC), podemos encontrar C' tal que $C' \equiv m^3 \pmod{N_1 N_2 N_3}$.

Assim, como

$$\begin{cases} m < N_1 \\ m < N_2 \\ m < N_3 \end{cases} \Rightarrow m^3 < N_1 N_2 N_3$$

Portanto, $C' = m^3 \Rightarrow m = \sqrt[3]{C'}$.

De forma mais geral, se os expoentes de encriptação forem todos iguais a e, pode-se recuperar M assim que se tenham $k > e$ onde k é o número de criptogramas interceptados. Portanto, este ataque só será bem sucedido se o expoente público e for relativamente pequeno.

7.2 Conclusão

É essencial, mesmo que facilite a encriptação, o uso de expoentes relativamente altos. Uma boa recomendação é o uso de no mínimo $2^{16}+1$, que, de certa forma, facilita esses cálculos e ainda é relativamente grande.

8 Ataque de Wiener

O ataque de Wiener é um método clássico de criptoanálise contra o RSA quando o expoente secreto d é anormalmente pequeno. Em 1990, Michael Wiener demonstrou que, se $d < \frac{1}{3}N^{1/4}$ então é possível recuperar d a partir do par público (N, e) utilizando frações contínuas, em tempo $O(\log N)$.

A ideia é que, para chaves fracas, a razão e/N possui convergentes que aproximam muito bem a razão $e/\varphi(N) \approx k/d$, permitindo recuperar d ao testar cada convergente (k, d).

8.1 Método

Seja (N, e) a chave pública. Calculamos a fração contínua de e/N, que é feito via o algoritmo estendido de Euclides. Para cada convergente (k, d) dessa fração, verificamos se $ed - 1$ é múltiplo de k, sugerindo que $\varphi(N) = \frac{ed-1}{k}$. Então testamos se essa $\varphi(N)$ leva a um par de fatores válidos (p, q) de N, resolvendo a equação quadrática $x^2 - (N - \varphi(N) + 1)x + N = 0$. Caso p e q sejam inteiros positivos e $pq = N$, então o valor correto de d foi encontrado.

```python
def wiener_attack(N, e):
    cf = continued_fraction(e, N)

    for (k, d) in convergents(cf):
        if k == 0: continue

        # Verify if (e*d - 1)/k is integer => possible phi(N)
        if (e*d - 1) % k != 0: continue

        phi_candidate = (e*d - 1) // k

        # Calculate s = p + q
        s = N - phi_candidate + 1
        disc = s*s - 4*N

        # Discriminant must be perfect square
        if not is_perfect_square(disc): continue

        t = isqrt(disc)
        p = (s + t) // 2
        q = (s - t) // 2

        # Verify p*q == N
        if p > 1 and q > 1 and p*q == N: return d  # Broke!

    return None  # Passed!
```

Listing 4: Ataque de Wiener

8.2 Conclusão

A segurança do RSA depende muito da escolha de um expoente secreto suficientemente "grande" (da ordem de $\varphi(N)$), pois quando isso não acontece, conseguimos aproximar $e/\varphi(N)$ por k/d e, como $\varphi(N)$ está muito próximo de N, também e/N. Geralmente, ainda é computacionalmente custoso determinar este d se ele for grande, mas dada a cota $d < \frac{1}{3}N^{1/4}$, essa aproximação se torna fina o bastante para satisfazer o critério clássico das frações contínuas, forçando k/d a aparecer nas convergentes da expansão de e/N, eliminando a necessidade de fatorar N.