

# Implementação de Gerador de Chaves RSA em Python

---

## 1 Introdução

Pretende-se neste relatório descrever a Implementação de uma função em python, `genRSAkey(1)` que receba como argumento um tamanho de chave  $\ell$  e retorne um tuplo da forma  $(n, p, q, e, d)$ , respeitando as condições descritas a seguir:

1.  $n = pq$  sendo  $p$  e  $q$  primos e por forma a que  $\log_2(n) \geq \ell$  (ou seja que  $n$  tenha na sua representação binária pelo menos  $\ell$  bits);
2.  $e$  inteiro coprimo com  $(p-1)(q-1)$ , ou seja, o máximo divisor comum entre  $e$  e  $(p-1)(q-1)$  seja 1;
3.  $d$ , por forma a que  $ed \equiv 1 \pmod{(p-1)(q-1)}$

## 2 Geração de candidatos a primos $p$ e $q$

Usando o módulo `random` do Python, a função `genRand(1)` gera números ímpares no intervalo  $2^{w-1} + 1$  e  $2^w - 1$  em que  $w = \lfloor \frac{\ell}{2} \rfloor$ , sendo que todos os números passíveis de serem gerados por `random.randrange` no intervalo descrito acima são depois **OR**-ed com 1 no seu bit menos significativo, aumentando assim a eficiência do gerador já que isto elimina a necessidade de gerar números candidatos a primos e depois testar num ciclo `while` a divisibilidade destes por 2, visto que qualquer primo  $> 2$  é necessariamente ímpar.

### 2.1 Crivo de Eratóstenes

Depois de gerado o número  $p$  (candidato a primo), verificar-se-à, numa primeira fase, contra um crivo de Eratóstenes, gerado apenas uma vez aquando a chamada de `genRSAkey(1)` e guardado em `small.primes`, contendo os 1000 primeiros números primos. Se algum destes é factor de  $p$ , então sabemos que  $p$  não é primo, sendo assim necessário invocar `genRand(1)` novamente e repetir o teste.

### 2.2 Teste de Miller-Rabin

No caso em que o número  $p$  gerado passa pelo processo de *sieving* supra descrito sem serem encontrados factores primos, prosseguimos para o teste de primalidade de Miller-Rabin (probabilístico). É de salientar que este último domina a complexidade temporal de `genRSAkey(1)`. Usar *sieving* é eficaz em minimizar o número de vezes em que é gerado um número composto  $p$  e este tem que ser sujeito ao teste de Miller-Rabin, ou seja: se  $p$  tem factores primos relativamente pequenos, detectamos desde logo que este é composto, gerando assim um novo  $p$  cuja primalidade volta a ser testada pelo mesmo processo, minimizando assim o número de iterações "desnecessárias" do teste de M-R. Formalmente, dada uma constante  $B$ , só serão sujeitos ao teste de M-R os candidatos a primos que se não se revelem *B-smooth* **TODO: INCLUIR FONTE SOBRE B-SMOOTH** e neste caso  $B$  é o último elemento do crivo de Eratóstenes.

O valor de `max_rounds` em `genRSAkey(1)`, corresponde ao número máximo de testemunhas aleatórias  $a$  geradas no intervalo  $[2, n-2]$  com relação à primalidade de um inteiro  $n$  ímpar tal que  $n > 3$ . Já que o teste de M-R retorna `False` quando  $n$  é composto, porque foi gerado um  $a$  que é testemunha da existência de factores primos de  $n$ , então podemos dizer que se  $n$  é um número composto, quanto mais iterações do teste de M-R forem feitas para esse  $n$ , maior é a probabilidade de encontrar uma testemunha  $a$  (gerada aleatoriamente) que respeite esses critérios e determine que  $n$  não é primo.

Baseando-nos no parágrafo anterior, podemos concluir que: se  $n$  é um número ímpar composto e ao fim de `max_rounds` o teste M-R não retornou **False**, então  $n$  é um "falso primo".

Sejam as variáveis aleatórias:

- $Y_k \leftarrow n$  é declarado primo depois de  $k$  iterações do teste de M-R;
- $X \leftarrow n$  é um número composto (sendo  $\overline{X} \leftarrow n$  é um número primo).

Por [1] e assumindo a veracidade da Hipótese Generalizada de Riemann é possível mostrar que pelo menos  $\frac{3}{4}$  das testemunhas aleatórias em  $a \in [2, n - 2]$  podem garantir que  $n$  é composto, que é o mesmo que dizer que no máximo  $\frac{1}{4}$  das testemunhas aleatórias no mesmo intervalo não garantem tal. Com isto podemos dizer que  $Pr[Y_k|X] \leq \frac{1}{4}$  para uma iteração do teste de M-R, ou seja  $k = 1$ . Sendo que para todas as  $k$  iterações do teste de M-R, é independente a escolha de  $a$ , mostrou-se em [2] que a probabilidade de erro deste teste pode ser descrita em função de  $k$  como  $Pr[X|Y_k] \leq (\frac{1}{4})^k$  e que  $Pr[Y_k|X]$  é relacionável com  $Pr[X|Y_k]$ , advindo do teorema de Bayes e conforme detalhado em [3].

Em vista dos resultados obtidos escolheu-se um  $k = 100$ , coincidente com `max_rounds` na função de geração de tuplos para as chaves RSA, o que admite uma probabilidade de erro igual ou inferior a  $\frac{1}{4^{100}}$ , algo admissível na geração dos factores primos de  $n$  (RSA *modulus*). Para suportar o argumento acima, de acordo com [4] (página 70) é dito que para uma probabilidade de erro até  $2^{-112}$ , são recomendadas pelo menos 56 iterações do teste M-R, quando  $p$  e  $q$  tiverem na sua representação 2048 bits.

### 2.3 Noção de *strong primes* em RSA

Apesar do módulo `random` na linguagem Python poder ser usado como função pseudo-aleatória para gerar grandes números candidatos a primos, é descrito na documentação [5] que este não é adequado para usos em aplicações criptográficas. Em vista dos possíveis riscos que surgem de nos afastarmos da verdadeira aleatoriedade

## 3 Escolha do expoente público

Como expoente público, decidimos escolher, de forma estática,  $e = 65537$ , uma vez que este número é usado de forma comum como expoente público no RSA.

A razão para ser tão comum deve-se a ser um número de Fermat ( $2^{2^n} + 1$ ), com  $n = 4$ ; um número primo suficientemente grande para evitar certos ataques ao RSA e a ter um peso de Hamming baixo (número de bits a 1), o que torna a sua computação em computadores binários extremamente rápida.

## 4 Cálculo de $d$

Para calcular o  $d$ , utilizamos uma extensão do algoritmo de Euclides, onde  $ax + by = \gcd(a, b)$ , uma vez que esta permite calcular um inverso modular multiplicativo, de forma muito rápida.

Da expressão  $ax + by = \gcd(a, b)$  ( $= 1$ , porque, neste caso,  $a$  e  $b$  são primos), pode conferir-se que  $ax \equiv 1(\text{mod}(b))$ , e da expressão  $ed \equiv 1(\text{mod}(p - 1)(q - 1))$ , pode simplesmente apurar-se que  $d = e^{-1}(\text{mod}(p - 1)(q - 1))$ , o que permite aplicar o algoritmo, para descobrir o  $d$ .

## 5 Tempos de execução e conclusões finais

### References

- [1] A. K. Lenstra, "Integer factoring," *Des. Codes Cryptography*, vol. 19, no. 2/3, pp. 101–128, 2000.

- [2] J.-M. Couveignes, T. Ezome, and R. Lercier, “A faster pseudo-primality test,” 2012.
- [3] G. Yarmish and J. Yarmish, “Finding large primes,” 2017.
- [4] F. I. P. Standards, “Digital signature standard (dss) - fips pub 186-4,” 2019.
- [5] P. S. Foundation, “Python language reference, version 3.” Disponível em: <https://docs.python.org/3/library/random.html>.