Implementação de Gerador de Chaves RSA em Python

1 Introdução

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

- $n = pq$ 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 ℓ bits);
- e inteiro coprimo com $(p-1)(q-1)$, ou seja, o máximo divisor comum entre e e $(p-1)(q-1)$ seja 1;
- d, por forma a que $ed \equiv 1(mod(p-1)(q-1))$

2 Geração de primos p e q

Usando o módulo `random` do Python, a função `genRand(l)` 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, visto que qualquer primo > 2 é necessariamente ímpar.

2.1 Crivo de Eratóstenes

Depois de gerado o número p (candidato a primo), verificar-se-á, numa primeira fase, contra um crivo de Eratóstenes, gerado apenas uma vez aquando da chamada de `genRSAkey(l)` 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(l)` 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(l)`. 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. Formalmente, dada uma constante B, só serão sujeitos ao teste de M-R os candidatos a primos que se não se revelem B-*smooth* [1] e neste caso B é o último elemento do crivo de Eratóstenes.

O valor de `max_rounds` em `genRSAkey(l)`, corresponde ao número máximo de testemunhas aleatórias a geradas no intervalo $[2, n-2]$ com relação à primalidade de um inteiro n, tal que $n > 2$. 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 aleatóriamente) que respeite esses critérios e determine que n não é primo.

Baseando-nos no parágrafo anterior, podemos concluir que: se n é um número 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] é possivel 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

Com base na hipótese de Riemann, o teorema dos números primos [5] implica que para um grande primo p, a sua proximidade com relação ao seu sucessor é, em média, $log(p)$ e $\lim_{p\to\infty} log(p) = \infty$. Isto significa que para números da ordem de 2048 bits, temos elevada probabilidade de gerar um par p e q que seja seguro no sentido em que n não seja facilmente factorizável, porque p e q não são suficientemente próximos. Mais concretamente, conforme provado em [6], devemos garantir que $|p - q| > 2^{\frac{k}{3}}$ (onde k é o número de bits na representação dos primos p e q) para que n não seja passível de ser factorizado em tempo polinomial, pelo método de de factorização de Fermat. A variável `safe_dist` na função apresentada é usada para a verificação de que os primos gerados satisfazem as condições supra descritas, tratando também o caso em que $p = q$.

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 extremamente rápida.

4 Cálculo de d

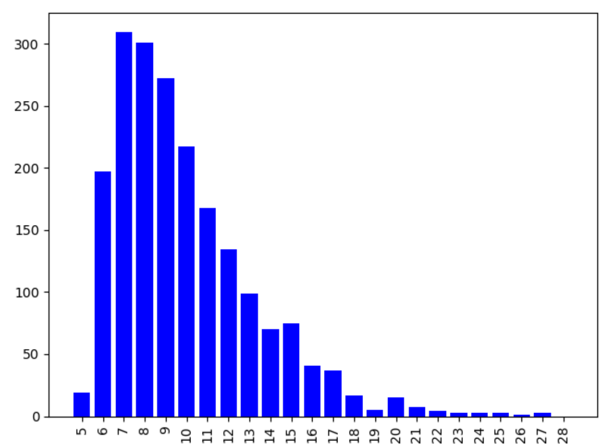
Para calcular o d, utilizamos uma extensão do algoritmo de Euclides, onde $ax + by = gcd(a, b)$, uma vez que esta permite calcular um inverso modular multiplicativo, de forma muito rápida.

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

5 Conclusões

Apesar de neste trabalho lidarmos com números de grande magnitude e isso mitigar de certa forma os riscos associados a ataques ao RSA baseados na factorização de n, achámos pertinente tomar certas medidas que descrevemos em cima quando justificável. É seguro afirmar que foi atingido o objectivo de geração de tuplos (n, p, q, e, d) em tempo que podemos considerar útil, sendo que no corpo de `genRSAkey(l)` existem instruções de controlo, por forma a garantir todas as condições referidas na Secção 1 e assegurar a correcção do algoritmo.

Abaixo inclui-se um histograma de frequências para os tempos de execução, na ordem dos segundos, de `genRSAkey(4096)` de um total de 2000 iterações, em que valores decimais foram arredondados para o inteiro mais próximo.



Da amostra referida de 2000 tempos de execução, podemos afirmar que o valor esperado do tempo de execução de `genRSAkey(l)`, para valores de ℓ na ordem de 2^12 (conforme esperado) é aproximadamente 9.94 segundos.

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] "Wikipedia - riemann hypothesis." https://en.wikipedia.org/wiki/Riemann_hypothesis#Large_prime_gap_conjecture.

[6] R. Erra and C. Grenier, "The fermat factorization method revisited," *IACR Cryptology ePrint Archive*, vol. 2009, p. 318, 2009.