

Segundo Trabalho Prático

Geração determinística de chaves RSA (D-RSA)

Implementação e Desenvolvimento

Neste relatório, é abordado o processo de implementação e desenvolvimento do projeto de criptografia com base nas especificações detalhadas enunciado do trabalho. Abordamos também a tradução de conceitos teóricos em código prático, explicando como cada componente é efetivamente implementado. Este trabalho foi desenvolvido nas linguagens C e Java, devido à flexibilidade e computação relativamente rápida comparativamente a outras linguagens de programação. Toda a documentação de ambas as implementações podem ser encontradas no ficheiro comprimido submetido anexado a este relatório.

Objetivos

- Implementar um **gerador de bytes pseudoaleatórios**, com uma **seed** com N bytes. A inicialização deste gerador será importante para contribuir para a geração do par de chaves RSA;
- Implementar um **gerador de chaves RSA**, que gere um par de chaves (pública e privada) com base num número primo de N bits, e que guarde as chaves em ficheiros de texto;
- Testar a performance do gerador de bytes pseudoaleatórios, em termos de tempo de execução.

Gerador de bytes pseudoaleatórios (*randgen*)

A configuração deste gerador deve ser longa e complexa, de modo a complicar a sua análise criptográfica.

- Parâmetros de Entrada
- Palavra-Passe;
- *Confusion string*;
- Contador de iterações;

Pseudocódigo

1. Computar uma *seed* para a palavra-passe, a *confusion string* e o contador de iterações, usando o método PBKDF2;
2. Computar uma *confusion pattern*, neste caso usaremos o mesmo método, que foi usado para computar a *seed*, pois a *confusion string* é dada no algoritmo PBKDF2 como o *salt*;
3. Inicializar o gerador com a *seed*;
4. Começar um ciclo com 'contador de iterações', iterações;
 - 4.1. Usar o PRNG, para gerar um conjunto de bytes pseudoaleatórios, até que o padrão da *confusion pattern* seja encontrado, como se vê na Fig.2;
 - 4.2. Usar o PRNG para produzir a nova *seed* e usar essa para gerar os bytes pseudoaleatórios.

Para o PRNG utilizado foi o SHA-256, onde é dada uma *seed* sendo gerado o *hash* desta, a cada iteração é gerado um novo *hash* usando o anterior como *seed*.



Figura 1-PBKDF2



Figura 2-Condição de paragem

Gerador de chaves RSA (*rsagen*)

O par de chaves RSA será gerado através do gerador de bytes pseudoaleatórios, da seguinte forma:

Pseudocódigo

1. Gerar um conjunto de bytes pseudoaleatórios com o tamanho da chave RSA (em bytes);
2. Será dividido o conjunto de bytes em dois conjuntos de bytes, sendo que o primeiro conjunto irá representar o nosso p e o segundo conjunto irá representar o nosso q , ambos em formato BIGNUMBER/BigInteger;
3. Será verificado se os dois conjuntos de bytes são primos, caso não sejam, o seu valor será incrementado até que sejam primos;
4. Será calculado o valor de n através da multiplicação $p * q$;
5. O valor de e é fixo, sendo que o seu valor é 65537 ($2^{16} + 1$);
6. Será calculado o valor de d através da função modular inverso de e e $\phi(n)$, sendo que este é calculado através da multiplicação $(p-1) * (q-1)$;

Armazenamento das chaves RRSA (*storekeys*)

Será guardado o par de chaves RSA em dois ficheiros de texto, sendo que um deles irá conter a chave pública (N , e) e o outro irá conter a chave privada (N , d).

Para armazenar ambas as chaves, foi usado o padrão DER (Distinguished Encoding Rules), sendo que o padrão DER é um formato binário codificado em base64 para ser guardado no ficheiro PEM.

Este padrão tem a seguinte estrutura:

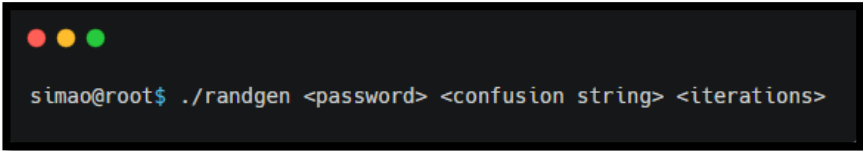
1. Byte de Início de Sequência (0x30): Indica o início de uma sequência de DER;
2. Tamanho Total da Sequência: O tamanho da sequência que inclui N e e ;
3. Byte de Separação para N (0x02): Indica o início de um número inteiro;
4. Tamanho do N : O tamanho do módulo N ;
5. O N (Módulo): Os bytes que representam o módulo.
6. Byte de Separação para e (0x02): Indica o início de outro número inteiro;
7. Tamanho do e ou d : O tamanho do expoente público e ou privado d ;
8. O e (expoente público): Os bytes que representam o expoente público.

Execução do programa

Em C

Para compilar o programa em C e Java, basta executar o *makefile*, através do comando *make*, e será criado um executável denominado *randgen*, outro *rsagen*, respetivamente para ambas as linguagens, e outro chamado *performance*.

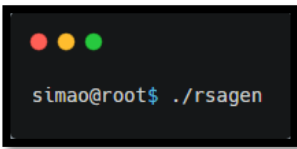
Para executar o programa *randgen* basta executar o seguinte comando:



```
simao@root$ ./randgen <password> <confusion string> <iterations>
```

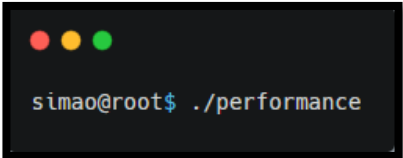
Nota: O resultado será retornado para o *stdout*.

Para executar o programa **rsagen** basta executar o seguinte comando:



```
simao@root$ ./rsagen
```

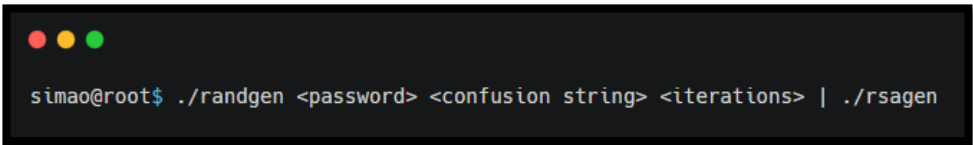
Para executar o programa *performance* basta executar o seguinte comando:



```
simao@root$ ./performance
```

Nota: A entrada do programa será feita através do *stdin*.

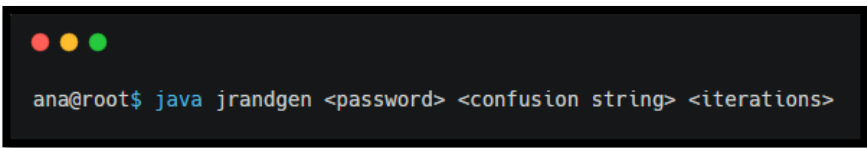
De modo a executar ambos utiliza o *pipe*, basta executar o seguinte comando:



```
simao@root$ ./randgen <password> <confusion string> <iterations> | ./rsagen
```

Em Java

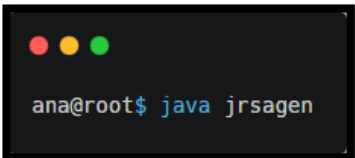
Para executar o programa *randgen*, basta executar o seguinte comando:



```
ana@root$ java jrandgen <password> <confusion string> <iterations>
```

Nota: O resultado será retornado para o *stdout*.

Para executar o programa *rsagen*, basta executar o seguinte comando:



```
ana@root$ java jrsagen
```

Nota: A entrada do programa será feita através do *stdin*.

De modo a executar ambos utiliza o *pipe*, basta executar o seguinte comando:

```
ana@root$ java jrandgen <password> <confusion string> <iterations> | java jrsagen
```

Testes de performance

Devido á grande computação que é feita pela aplicação *randgen*, apenas foi testado a performance do *setup* do gerador de bytes pseudoaleatórios.

Em C

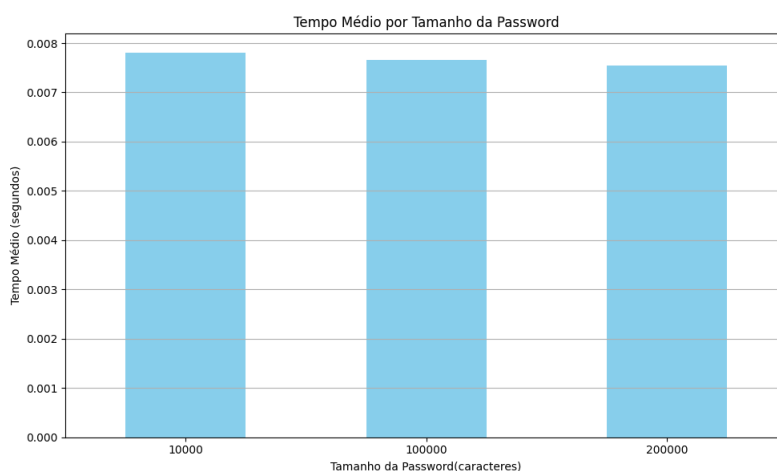


Figura 3-Tempo Médio por tamanho da password em C

No gráfico acima, observamos que o tamanho da palavra-passe não tem um impacto significativo no tempo médio de execução. Independentemente do tamanho da palavra-passe, o tempo médio de execução não varia muito para as diferentes combinações de tamanho do *salt* e iterações.

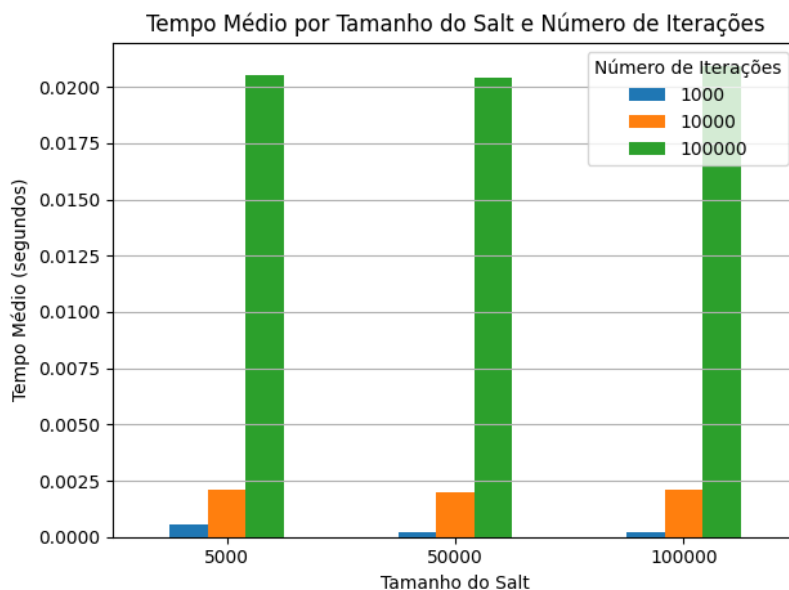


Figura 4-Tempo Médio por Tamanho do Salt e número de iterações em C

No gráfico acima, podemos observar que, para os dados fornecidos, o tamanho do *salt* parece ter uma influência limitada no tempo médio de execução. Não há uma tendência clara de aumento ou diminuição consistente no tempo médio à medida que o tamanho do *salt* muda para os

valores apresentados.

No entanto, o número de iterações tem uma influência mais evidente no tempo médio de execução. Conforme o número de iterações aumenta, há um aumento notável no tempo médio necessário para executar o algoritmo.

Em Java

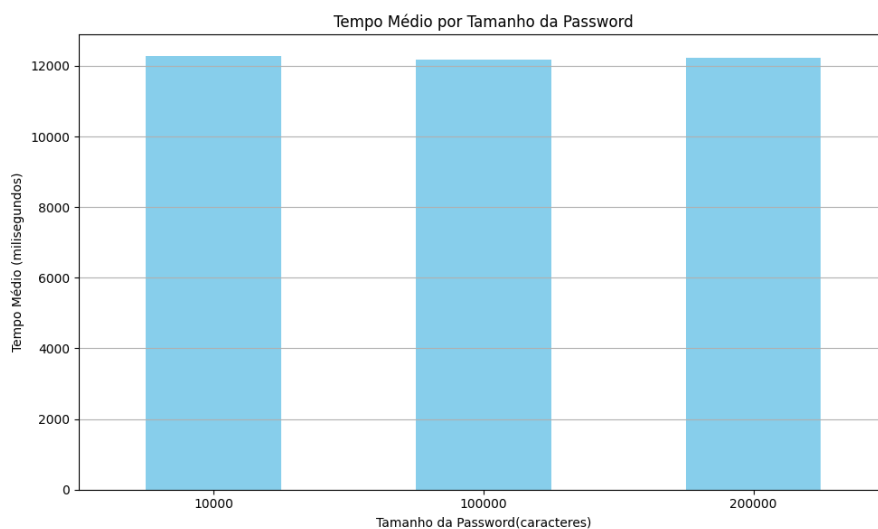


Figura 5-Tempo Médio por tamanho da password em Java

No gráfico acima, revela que não há variação aparente no tempo de execução em relação ao tamanho da palavra-passe. Ou seja, o tamanho da palavra-passe não parece impactar significativamente o tempo de execução do algoritmo PBKDF2.

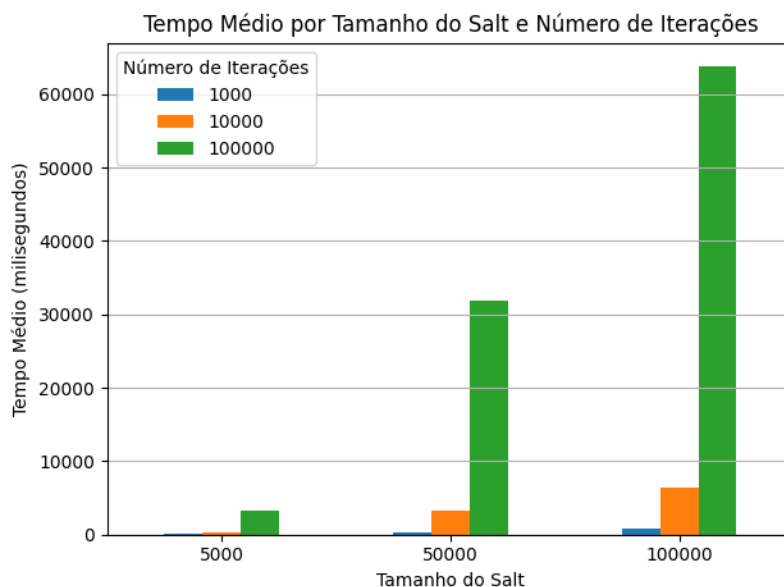


Figura 6-Tempo Médio por Tamanho do Salt e número de iterações em Java

No gráfico acima, podemos observar que à medida que o tamanho do salt e o número de iterações aumentam, o tempo médio de execução também aumenta. É evidente que tanto o tamanho do salt quanto o número de iterações influenciam significativamente no tempo necessário para executar o algoritmo PBKDF2. Aumentar o tamanho do salt ou o número de iterações resulta em um tempo de execução maior, conforme demonstrado pelos cálculos das médias.

Conclusão

Em suma, a implementação da geração de chaves RSA foi desafiadora, porém enriquecedora, permitindo a tradução eficaz de conceitos teóricos em soluções práticas. A escolha das linguagens revelou-se acertada, garantindo flexibilidade e eficiência computacionais essenciais para o sucesso do projeto.

Através dos testes de desempenho do *setup*, podemos chegar à conclusão que a complexidade da computação existente no PBKF2 encontra-se no número de iterações e tamanho de bytes a gerar, sendo que o tamanho da palavra-passe em nada influencia na sua complexidade computacional.

Este trabalho não apenas cumpriu os nossos objetivos, mas também proporcionou uma aplicação prática significativa dos conhecimentos teóricos lecionados na unidade curricular. O comprometimento com a qualidade e a seleção estratégica de ferramentas resultaram na realização bem-sucedida desta iniciativa de criptografia.

Referências

- CryptoSys. (23 de 11 de 2023). Obtido de RSA Key Formats:
<https://www.cryptosys.net/pki/rsakeyformats.html>
- IBM. (23 de 11 de 2023). Obtido de Size considerations for public and private keys:
<https://www.ibm.com/docs/en/zos/2.3.0?topic=certificates-size-considerations-public-private-keys>
- OpenSSL Project. (23 de 11 de 2023). Obtido de OpenSSL: <https://www.openssl.org/>
- Oracle. (23 de 11 de 2023). Obtido de Oracle Documentation:
<https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html>
- Wikipedia. (23 de 11 de 2023). Obtido de Wikipedia PBKDF2:
<https://en.wikipedia.org/wiki/PBKDF2>
- Wikipedia. (23 de 11 de 2023). Obtido de Wikipedia PKCS by RSA Security LLC:
<https://en.wikipedia.org/wiki/PKCS>