DRSA

Criptografia Aplicada

Mestrado em Cibersegurança

Sofia Teixeira Vaz
(92968) sofiateixeiravaz@ua.pt

20/01/2022

Conteúdo

Capítulo 1

Introdução

Este documento explicita como a implementação de um sistema de geração de pares de chaves de Rivest-Shamir-Adleman cryptosystem (RSA) foi feita, apelidada Deterministic RSA (D-RSA). O capítulo presente visa a explicitar alguns termos e requisitos do sistema criptográfico presente.

1.1 RSA

RSA é um sistema de chaves assimétricas. Isto significa que, para cifrar e decifrar mensagens, são usadas chaves diferentes.

No caso do RSA, cada interveniente terá a sua chave pública, usada para cifrar os dados, e a chave privada, usada para os decifrar.

Este sistema tem alguns valores associados.

Primeiramente, são precisos dois valores primos, p e q. Estes primos nunca deverão ser reutilizados, e $p-1$ e $q-1$ não deverão ter fatores primos pequenos. Destes dois valores vem N, que será $N = p * q$.

Também é necessário definir e, que integrará a chave pública. Há valores vistos como aceitáveis para e.

Com e será possível calcular d, que será igual a $d = e^{-1} \bmod \lambda(N)$.

De notar que $\lambda(N) = lcm(p-1, q-1)$.

A chave pública será, então, formada por N e e, e a mensagem será cifrada ao calcular:

$$C = M^d \bmod N \tag{1.1}$$

M é a mensagem decifrada, e C a mensagem cifrada. Tanto d como N são os valores da chave pública do interveniente para o qual a mensagem é destinada.

A chave privada será formada por N e d, e é usada para decifrar a mensagem enviada (C) mencionada acima. Esta é gerada calculando:

$$M = C^e \bmod N \tag{1.2}$$

RSA também é usado para confirmar a identidade de quem tenha enviado a mensagem, mas esse modo de uso não será discutido no documento.

1.2 D-RSA

O D-RSA é, em suma, um gerador de pares de chaves RSA.

O sistema tem um gerador de pares de chaves RSA e um gerador aleatório de números determinístico.

Este gerador de valores aleatórios requer uma password, *confusion string* e número de iterações.

Inicialmente, será criada uma *seed* através dos três valores mencionados acima.

Depois, através da *confusion string*, será gerada uma sequência de bytes com o mesmo comprimento, o *confusion pattern*.

Assim, o gerador será inicializado com a *seed* e deverá gerar bytes até o *confusion pattern* estar presente neles. Este passo deverá ser repetido tantas vezes quanto o número de iterações, e em cada ciclo a *seed* será gerada pelo próprio gerador, antes deste ser reinicializado.

Também existem duas aplicações.

Uma delas é um gerador de bytes aleatório, que utiliza o gerador acima mencionado. Esta mesma aplicação também tem modo de utilização *speed*, que verifica o tempo que o gerador demora a ser inicializado tendo em conta os vários valores de entrada.

A segunda aplicação é um gerador de pares de chaves RSA, que utiliza o *stdin* para obter os valores p e q. De notar que a aplicação também verifica e gera valores apropriados para RSA tendo em conta as regras mencionadas acima. Com isto, é possível ter qualquer processo como *input* da aplicação, como por exemplo uma leitura de `/dev/urandom` ou a própria aplicação geradora de *bytes* aleatórios.

Capítulo 2

Implementação

O sistema foi implementado em Python, e faz uso de algumas bibliotecas. A biblioteca `math` foi usada para alguns cálculos. A biblioteca `time` é usada para medir intervalos de tempo, e `matplotlib` é usada para gerar gráficos. Também foi usada a biblioteca `json` para gerar os ficheiros das chaves, e a biblioteca `sympy` é usada para encontrar primos. Finalmente, a biblioteca `sys` é usada para melhor controlo do fluxo do programa, e `hashlib` é usada para *hashing*.

2.1 Gerador aleatório

O gerador segue a sequência de instruções mencionada acima. De notar que as especificações acima não têm em conta, no entanto, como a *seed*, o *confusion pattern* e a *stream* serão geradas.

Para gerar a *seed*, a *password* e a *confusion string* são somadas, e isto será *hashed* em MD-5 tantas vezes quantas iterações foram definidas.

Para definir o *confusion pattern*, este será gerado transformando a *string* em *bytes*, usando um dos *encodings* definidos na lista.

No que toca à geração de valores aleatórios, o processo é relativamente simples. Existem três listas: a das posições, que será inicialmente um *array* de 0 a 4, a lista de funções de *shuffle* e a lista de funções alteradoras. Para gerar a *stream*, é removido o primeiro valor da lista de posições. Este valor é colocado de novo na lista, mas desta vez no fundo. Tendo a posição, esta definirá a função de *shuffle* e de alteração. Ambas as funções são removidas e colocadas no fim das listas correspondentes. A função de *shuffle* é aplicada à *seed* atual, e depois disso a alteradora também é aplicada.

Dependendo do número de *streams* geradas pelo mesmo gerador, as operações e a ordem destas será diferente. Considerando que são geradas *streams* até o *confusion pattern* ser encontrado, esta variabilidade será notável. Assim, é criado um nível de entropia desejável, e o gerador em si será mais pesado em termos de memória.

As funções de *shuffle* são as seguintes:

- shuffle_using_seed
- shuffle_using_self
- shuffle_using_positions

De notar que, após aplicar uma função de *shuffle*, é possível que nem todos os valores do estado inicial estejam presentes.

As funções alteradoras são as seguintes:

- sum_with_seed
- subtraction_with_seed
- sum_with_self
- subtraction_with_self
- xor_with_seed
- multiply_with_seed
- multiply_with_self
- add_with_seed_string_enc
- add_with_self_string_enc
- hash_self

Este gerador está presente em `srand.py`, sendo a classe do gerador descrito a `random_generator`.

2.2 RSA

A implementação do RSA é simples, recebendo como *input* os valores p e q. A partir daí gera os valores N, e e d, não verificando se o *input* constitui bons valores.

2.3 randgen.py

A aplicação `randgen.py` tem dois modos de funcionamento.

O modo de geração contínua requer execução na seguinte forma:

```
python3 randgen.py PASSWORD CONFUSION_STRING ITERATIONS
```

Em `PASSWORD` deverá ficar a password a ser usada na geração, em `CONFUSION_STRING` ficará a *confusion string*, e em `ITERATIONS` o número de iterações.

O modo *speed* requer que o *script* seja iniciado usando um comando parecido com `python3 randgen.py speed REPORT_FILE`, onde `REPORT_FILE` será o

nome do ficheiro no qual os dados de execução ficarão. A execução neste modo gera o relatório com dados de cada execução e, para consulta mais fácil, os dados dos casos que demoraram mais e menos tempo ficarão nas primeiras linhas. Também é gerado um gráfico, que ficará guardado no ficheiro `graphs.png`.

A execução poderá ser parada a qualquer momento pelo utilizador, sendo os valores obtidos até ao momento registados nos ficheiros mencionados acima.

2.3.1 Resultados do randgen.py

Durante testes desta aplicação, houve várias observações interessantes.

Primeiramente, uma *confusion string* maior significa sempre maiores tempos de inicialização para o gerador, ao ponto de deixarem de haver recursos suficientes.

Um exemplo de execução para um variável tamanho da *confusion string* é visível em Figura 2.1.

Este gráfico também traz algumas conclusões importantes. O caso no qual o tempo de execução é mais alto, a *password* é menor, e o número de iterações maior. Isto faz sentido, uma vez que um maior número de iterações significa um maior número de vezes que o padrão deve ser encontrado, e uma *password* menor significa uma *seed* (e, assim, *stream*) menores. Com *streams* menores, será menos provável o padrão estar presente, logo, será necessário criar *streams*, que a cada iteração crescem, mais vezes.

No que toca aos contributos da *iteration count* e *password*, um *snippet* do resultado da execução pode ser visto abaixo. De notar que nesta execução o tamanho da *confusion string* foi fixo em 1, enquanto que a *password* variava de tamanho entre 1 e 9, e a *iteration count* variava, também, entre 1 e 9.

```
LONGEST TIME:
        TIME: 7.82012939453125e-05seconds; passwd: 8; iter count 1; conf string: 1
SHORTEST TIME:
        TIME: 0.06823396682739258seconds; passwd: 9; iter count 7; conf string: 1
```

Como é visível, demorou mais quando tanto a *password* e *iteration count* foram maiores. Isto faz sentido, uma vez que uma *password* mais longa demorará mais tempo a ser processada, e considerando a *iteration count* maior, esta *password* será processada, necessariamente, mais vezes.

A alternativa que demorou menos tempo é aquela na qual a *password* é maior, e a *iteration count* menor. Isto também é expectável: com uma *password* maior, será mais fácil encontrar o *confusion pattern*, e requerindo menos iterações, este processo de procura, *shuffling* e geração será, necessariamente, feito menos vezes.

2.4 rsagen.py

Esta aplicação gera um par de chaves RSA usando os valores inseridos no *stdin*, sobre a assunção que este terá um gerador de bytes aleatórios associado. Assim,

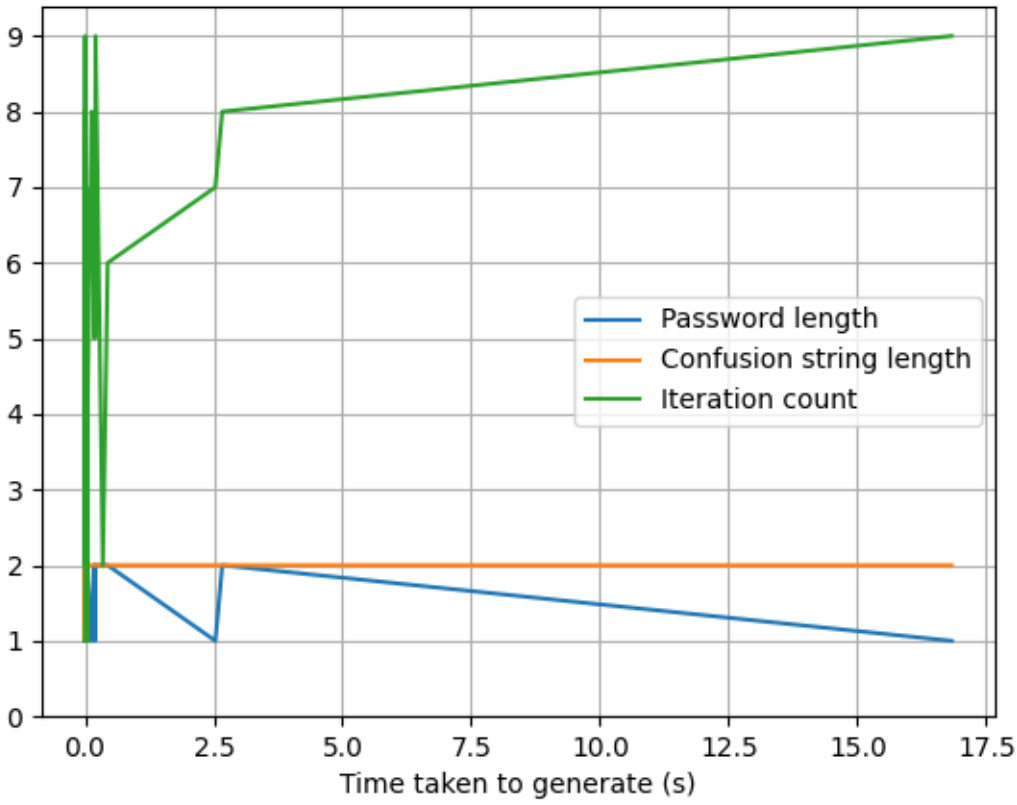

Figura 2.1: Tempo de execução em função dos três parâmetros

será possível usar a aplicação `randgen.py` como fonte de valores. Para o fazer, o comando usado para iniciar a aplicação deverá ser:

```
python3 randgen.py PASSWORD CONFUSION_STRING ITERATIONS | python3 rsagen.py
```

No final da execução, as chaves estarão guardadas em dois ficheiros - `public.json` e `private.json`. Como o nome sugere, as chaves são guardadas em formato JSON.