



O Sistema Merkle-Hellman Knapsack

Bernardo Rodrigues
a79008@alunos.uminho.pt

César Silva
a77518@alunos.uminho.pt

Maria Francisca Fernandes
a72450@alunos.uminho.pt

Universidade do Minho — 6 de Maio de 2019

Resumo

Este documento apresenta os vários passos e considerações feitas para implementação do sistema em questão. Assim como, alguns factos relativos a este.

Conteúdo

1	Introdução	3
2	Implementação	4
2.1	Geração de Permutações	4
2.2	Geração de uma sequência super crescente aleatória	4
2.3	Geração de Coprimos	5
2.4	Geração das Chaves	6
2.4.1	Geração da chave Pública	6
2.4.2	Geração da chave Privada	6
2.4.3	Versão Multi-Iterada	7
2.5	Encriptação	7
2.6	Deciptação	7
2.6.1	Versão Multi-Iterada	7
3	Conclusões	9

Lista de Algoritmos

1	Geração da sequência super crescente aleatória	5
2	Gerador de Coprimos - brute force	6

Capítulo 1

Introdução

Este trabalho foi desenvolvido no âmbito da Unidade Curricular de *Teoria de Números Computacional*. De entre as escolhas possíveis, foi escolhido estudar o sistema *Merkle-Hellman Knapsack*.

Este foi um dos pioneiros da criptografia de chave pública, inventado por **Ralph Merkle** e por **Martin Hellman** em 1978, 2 anos após a introdução deste conceito. Ao contrário do **RSA**, um outro sistema de chave pública, este não serve para fazer autenticações a partir de assinaturas digitais. Devido à encriptação deste só usar a chave pública e a deciptação só usar a chave privada. O sistema, assenta num problema *subset sub problem* um caso especial do problema da *Mochila*, conhecido por ser **NP-complete**. Utiliza uma variante "fácil" que acenta em sequências super crescentes para as chaves privadas, e uma "difícil" para as chaves públicas.

Capítulo 2

Implementação

Link da implementação no CoCalc - aqui - Kernel usado: **SageMath**, Py3.

Ao longo das secções deste capítulo apresentamos os vários algoritmos seguidos para a codificação do sistema. A implementação destes segue nos ficheiros de texto que acompanham o relatório.

2.1 Geração de Permutações

Um dos passos da Geração da Chave Pública - 2.4 - consiste em gerar uma permutação de uma sequência. Para tal utilizamos o algoritmo proposto por **Sandra Sattolo**[1].

Este itera uma lista - seq - a partir do último índice desta - n . Em cada passo calculamos um índice aleatório - j - tal que $1 \leq j < n$ e de seguida trocamos os valores de seq_j e seq_n . Finalmente, terminámos quando $n = 1$.

Na nossa implementação usamos listas para guardar as permutações.

Na implementação, a função tem o nome **sattoloShuffle**.

2.2 Geração de uma sequência super crescente aleatória

Um dos componentes da Chave Privada - 2.4 - é uma sequência, esta é considerada super crescente se:

Definição 1. Consideremos uma sequência de números b_1, \dots, b_n . Esta diz-se super crescente se:

$$b_i > \sum_{j=1}^{i-1} b_j \text{ para cada } i, 2 \leq i \leq n.$$

Como tal, conseguimos deduzir:

$$b_1 + b_2 + \dots + b_k < 2 \times b_k$$

Ou seja, precisamos apenas de considerar o último valor gerado para calcular um próximo. Com isto apresentamos o nosso algoritmo.

Algoritmo 1 Geração da sequência super crescente aleatória

Recebe: n - o tamanho da sequência

Devolve: $\{x_1, \dots, x_n\}$ - uma sequência super crescente aleatória

- 1: k um limite aleatoriamente grande
 - 2: f uma função que satisfaz $f(x) > 2 * x$
 - 3: $x_1 \leftarrow j$ tal que $1 \leq j \leq k$ aleatório
 - 4: **for** x_i com $i := 2$ **até** n **do**
 - 5: $x_i = f(x_{i-1})$
 - 6: **end for**
-

Na implementação, a função tem o nome **randomSupIncSequence**.

2.3 Geração de Coprimos

Também durante a Geração das Chaves - 2.4 - do sistema temos de considerar dois valores, M e W . Dada uma sequência super crescente $\{b_1, \dots, b_n\}$, M verifica:

$$M > \sum_{j=1}^N b_j$$

W por sua vez, verifica:

$$1 \leq W < M \text{ e } \gcd(W, M) = 1$$

O algoritmo considerado adopta uma estratégia brute force. Isto é possível face ao discutido num artigo de *StackExchange* [2], que demonstra que calcular o gcd é uma operação não custosa. Este considera um valor aleatório para W compreendido no intervalo apresentado acima, testando o gcd de este com M . Se forem coprimos o processo para, caso contrário somamos um a este e repetimos o teste.

Algoritmo 2 Gerador de Coprimos - brute force

Recebe: M - um número

Devolve: W - um número coprimo com o argumento, $1 \leq W < M$

```
1:  $W \leftarrow$  número aleatório tal que  $1 \leq W < M$ 
2: while  $\gcd(M, W) \neq 1$  do
3:    $W = W + 1$ 
4: end while
```

Na implementação, a função que calcula um M tem o nome **findM**. Na implementação, a função que calcula um W tem o nome **findW**.

2.4 Geração das Chaves

Reunindo os vários algoritmos decritos nas secções anteriores, podemos agora construir as Chaves do sistema. Consideremos as várias componentes precisas ao longo desta secção.

- Uma permutação - π - 2.1
- Uma sequência super crescente aleatória - $\{b_1, \dots, b_n\}$ - 2.2
- Dois inteiros coprimos W e M - 2.3. Para as versões multi iteradas ainda extendemos estes valores para 2 sequências $\{W_1, \dots, W_n\}$ e $\{M_1, \dots, M_n\}$ tais que $\gcd(W_i, M_i) = 1$ para $i = 1$ até n .

O algoritmo seguido é dado na bibliografia recomendada [3]. Com isto, a geração da chave pode ser dividida em duas subcategorias.

2.4.1 Geração da chave Pública

Definição 2. São computados a_i tal que:

$$a_i = W \times b_{\pi_i} \pmod{M} \text{ para cada } i \text{ tal que } 1 \leq i \leq n.$$

A chave publica é dada por (a_1, a_2, \dots, a_n)

2.4.2 Geração da chave Privada

A chave privada é dada por $(\pi, M, W, (b_1, b_2, b_3, \dots, b_n))$

2.4.3 Versão Multi-Iterada

Uma variante do algoritmo base de Merkle-Hellman envolve disfarçar a sequência super crescente, através de sucessivas multiplicações modulares.

Neste caso precisamos de fixar outro inteiro, T , que ditará o numero de iterações para geração da chave. Mais uma vez um inteiro N é fixo como parâmetro de sistema.

Para $1 \leq j \leq T$ são feitos os cálculos descritos em 2.3. Para calcular a sequência atual são calculados M_j e W_j , da mesma maneira que eram calculados na geração da chaves no algoritmo básico. O M_j é calculado com base na sequência super crescente anterior (índice $j - 1$) sendo depois calculado W_j de maneira igual ao calculado na versão normal. A sequência dada na primeira iteração a_1, \dots, a_n é a que será usada na Chave Privada.

Na implementação, a função que gera as Chave Simples tem o nome **KSKeyGen**. Para a versão Multi Iterada a função tem o nome **MIKS-KeyGen**.

2.5 Encriptação

A encriptação é feita a partir da chave pública a_1, a_2, \dots, a_n . É representada a mensagem m a encriptar em binário como uma string de tamanho N . ($m = m_1 m_2 \dots m_n$)

Finalmente é calculado um inteiro c tal que: $c = \sum_{i=1}^N a_i \times m_i$.

Na implementação, a função da versão simples tem o nome **KSEncrypt**. A versão multi iterada tem o nome **MIKSEncrypt**

2.6 Decriptação

Para recuperar a mensagem m a partir de c , o dono da chave privada deve: Calcular $d = W^{-1} \times c \pmod{M}$.

Encontrar inteiros $r_1, \dots, r_n, r_i \in \{0, 1\}$ tal que $d = r_1 \times b_1 + r_2 \times b_2 + \dots + r_n \times b_n$. Os bits da mensagem são então $m_i = r_{\pi_i}, i = 1, 2, \dots, n$.

2.6.1 Versão Multi-Iterada

A decriptação desta variante do algoritmo é análoga à sua encriptação, ou seja, os cálculos efetuados são os mesmos do que na versão básica do algoritmo, mas são efetuados T vezes.

A decriptação é feita a partir de sucessivamente calcularmos:

$$d_j = W^{-1} \times d_{j+1} \pmod{M_j}, \text{ para } j = t, t-1, \dots, 1$$

onde $d_{t+1} = c$.

Finalmente só temos de encontrar inteiros $r_1, \dots, r_n, r_i \in \{0, 1\}$ tal que:

$$d_1 = r_1 \times a_1 + r_2 \times a_2 + \dots + r_n \times a_n.$$

E os bits da mensagem são recuperados a partir de aplicarmos a permutação π .

Na implementação, a função para a versão normal tem o nome **KSDencrypt**. A versão multi iterada tem o nome **MIKSDecrypt**.

Capítulo 3

Conclusões

Após a realização do trabalho e depois de compreendido o problema em questão foi possível concluir que o sistema **Merkle-Hellman Knapsack** não é seguro, tal como ficou provado, apenas alguns anos após a sua publicação, por **Adi Shamir**. Existem várias maneiras de o atacar, de entre estas, destaca-se o **Lenstra-Lenstra-Lovász**[3], que reduz o problema *subset sum problem* a um de encontrar um *short vector in a lattice*, baseada em aproximações, que resolve o problema em tempo polinomial. Deixamos como trabalho futuro uma possível implementação deste ataque.

Bibliografia

- [1] Wikipedia, *Fisher–Yates shuffle* — *Wikipedia, The Free Encyclopedia*, <http://en.wikipedia.org/w/index.php?title=Fisher%E2%80%93Yates%20shuffle&oldid=882337194>, Acedido em 05-May-2019.
- [2] *Finding a coprime of a general magnitude*, <https://math.stackexchange.com/questions/2430742/finding-a-coprime-of-a-general-magnitude>.
- [3] A. J. Menezes, P. C. van Oorschot e S. A. Vanstone, *Handbook of applied cryptography*. CRC Press, 1997, Acedido a partir de <http://cacr.uwaterloo.ca/hac/about/chap8.pdf>, ISBN: 978-0849385230.