Universidade de Aveiro

Merkle-Hellman cryptosystem

Tiago Portugal 103931

Afonso Azevedo

David Cobbilac

Algoritmos e estruturas de dados - 40437 1º Semestre DETI

31/12/2021

Introdução

O sistema de enciptação Merkel-Hellman knapsack foi um dos primeiros sistemas de encriptação de chave publica a ser inventados sendo criado por Ralph Merkle and Martin Hellman em 1978.
No entanto apesar do metodo ser mais simples que as normas establecidas anteriormente, foram rapidamente encontradas falhas no algoritmo.

O metodo de encriptação deste algoritmo é assimétrico pelo que gera duas chaves, uma **publica** e outra **privada**.

Ex:

> O Tiburcio e Gertruncia querem partilhar um documento confidencial e para isso vão usar o sistema de encriptação Merkle and Hellman.
>
> O sistema de encriptação vai gerar a cada um, um par de chaves, uma privada que não devem partilhar e outra publica para proceder á encriptação.
> O documento vai ser enviado da Gertruncia para o Tiburcio, então para isso , primeiro a Gertruncia terá que pedir a chave publica do Tiburcio.
>
> Após receber a chave a Gertruncia terá que fazer uma copia do ficheiro e encriptar essa copia através do algoritmo de Merkel-Hellman. Esta cópia é necessária pois após ser feita a encriptação com a chave publica do Tiburcio só este a pode desencriptar o documento com a sua chave privada.
>
> No final ela terá de enviar o documento ao Tiburcio que á chegada, ele usará a sua chave privada para proceder á desencriptação.

Encriptação de uma palavra de n bits

Neste trabalho vamos explorar uma verão simplificada deste algoritmo onde o problema de knapsack será substituido por um problema de somas de um subconjunto.

Geração de chaves

A chave privada consiste em 3 elementos:

> **Super sequencia gerada aleatóriamente.**
>
> $$W = \{w_1, w_2, w_3, \ \dots, w_n\}$$
>
> $$\text{onde}$$
>
> $$w_{i+1} > \sum_{i=0}^{n} w_i$$
>
> **Constante *m* aleatória**
>
> $$m > \sum_{i=0}^{n} wi$$
>
> **Constante *a* aleatória coprima de *m***
>
> $$mdc(m, a) = 1$$

A chave publica **P** que consiste num array de inteiros

> $$P = \{p_1, p_2, p_3, \ \dots, p_n\}$$
>
> $$\text{que obtemos ordenando}$$
>
> $$W = \{w_1, w_2, w_3, \ \dots, w_n\}$$
>
> $$\text{onde}$$
>
> $$w'_i = a * w_i \bmod m$$

Encriptação

Dado um array **B** de **n** bits o resultado da encriptação será um numero inteiro **C'** tal que:

$$C' = \sum_{i=0}^{n} b[i] * p[i]$$

Desencriptação

Numa situação normal, tendo acesso á chave privada usariamos uma algoritmo euclidiano em com um algoritmo ambisioso "greedy" para eficientemente resolvermos um problema de subsomas para chegar aos dados originais.

No entanto neste projeto não nos é dada á chave privada, sendo o nosso objétivo criar e explorar as possiveis formas de apartir da chave publica chegar á privada documentando e estudando as complexidades computacionais (temporais e espaciais) dos diferentes metodos.

Brute_Force

Este implementação consiste em percorrer todas as somas possiveis iterativamente até encontarmos a soma correta.
Dada uma chave publica de **n** elementos o conjunto de subsomas possiveis a percorrer será **2^n**, pelo que ,teoricamente, a **Big O notation será 2^n*n** sendo o ultimo n relativo ao ciclo for usado para confirmar a soma.

```
int brute_force_recursive(int n, integer_t p[n], integer_t
desired_sum, int idx, integer_t partial_sum, int b[n]){
// verifica valida cada bit dos dados encriptados comforme a soma
parcial atá chegar á soma desejada
        int i;
        if (partial_sum == desired_sum){
                for (i = idx; i < n; i++){
                        b[i] = 0; // mete o resto da chave a zero
                }
                return 1;
        }

        if (idx == n || partial_sum > desired_sum){
                return 0;
        }
```

```
        b[idx] = 0;
        int r = brute_force_recursive(n, p, desired_sum, idx + 1,
partial_sum /* + p[idx]*b[idx] */, b);

        if (r != 0){
                return r;
        }

        b[idx] = 1;
        r = brute_force_recursive(n, p, desired_sum, idx + 1,
partial_sum + p[idx] /* + p[idx]*b[idx] */, b) != 0;
        return r;
}
```

```
int brute_force_iterative(int n,integer_t p[n],integer_t
desired_sum,int b[n]){

        long long int total_sums = 1 << n;
        long long int i,j,sum;

        for(i = 0; i<total_sums;i++){
                sum = 0;
                for(j = 0; j<n ;j++){
                        if (i & (1 << j)){
                                b[j] = 1;
                                sum += p[j];
                        }
                }

        if(sum == desired_sum){
                return 0;
        }

        memset(b, 0, n*sizeof(int));

        }

        return 1;
}
```

Implementamos este metodo de duas formas, **recursivamente e iterativamente**, quando aplicado em geral o **metodo recursivo** quando verificando exprimentalmente é ligeiramente mais eficiente do que o método recursivo.

[[graph here]]

Horowitz and Sahni

Baseada no metodo desenvolvido por Ellis Horowitz e Sartaj Sahni para o problema de subsomas, esta implementação ao custo da memória acaba por ser mais efeciente que o **brute force** em termos temporais.

Aqui diminuimos o numero de somas a ser calculadas para **2*2^(n/2)**, dividindo o array de forma relativamente parcial em duas partes, calculando as suas subsomas. Apartir das subsomas por um **metodo de ponteiros ascendente e descendente** sumando os dois valores dos ponteiros até chegarmos a soma desejada (ponteiro ascendente → 1º array, ponteiro descendente → 2º array), com o valor desses ponteiros aplicamos o anterior método **brute force** (recursivo) com o **array de subsomas, subarray e a subsoma** chegando assim á mensagem original.

```
void hs(int n,integer_t p[n],integer_t desired_sum,int b[n]){

        int n1 = n/2;
        int n2 = n - n1;
        integer_t t=1ULL<<n1;
        integer_t* lower = (integer_t*)malloc(sizeof(integer_t)*t);
        t=1ULL<<n2;
        integer_t* upper = (integer_t*)malloc(sizeof(integer_t)*t);
        integer_t i = 0;
        integer_t j = (1ULL << n2) -1;

        make_sums(n1,p,lower);
        make_sums(n2,p+n1,upper); // função auxiliar no fim do
documento

        while(1){
                if(upper[j]+lower[i] == desired_sum){
                        break;
                } else if (upper[j]+lower[i] < desired_sum){
                        i++;
                        continue;
                }
                j--;
        }

        brute_force_recursive(n1,p,lower[i],0,0,b);
        brute_force_recursive(n2,p+n1,upper[j],0,0,b+n1);
        free(lower);
        free(upper);
}
```

Em termos de complexidade computacional temporal este algorimo terá **Big O notation** de **4∗2^(n/2)*n/2** (4 → 2*O(n) de brute force) , ignorando as constantes , **2^(n/2)*n/2**, que se confirma exprimentalmente.

E em termos de complexidade computacional espacial este algoritmo terá **Big O notation** de **(2^n/2 ∗2)**.

[[Insert graph here]]

Schroeppel and Shamir

Este método é basedo num melhoramente feito por **Schroeppel and Shamir** que visa melhorar a complexidade temporal chegando á soma desejada após dividir a chave publica e calcular as suas somas, iterativamente.

Para isso ultilizamos uma priority queue que ...