Acordo de chaves de Diffie-Hellman

Ranieri S. Althoff[1]

[1]Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Segurança em Computação

1. Introdução

O acordo de chaves de Diffie-Hellman (D-H) é um método de troca de chaves criptográficas em meio público (desprotegido) e é um dos primeiros protocolos de chave pública no meio da criptografia.

Tradicionalmente, a comunicação criptografada entre duas pessoas exigia que elas primeiro escolhessem uma chave e a transmitissem por um meio seguro, como um papel transportado por um mensageiro confiável, mas o D-H permite que os lados da comunicação não necessitem se conhecer para estabelecer uma chave secreta sob um canal inseguro.

2. Definição

O acordo de Diffie-Hellman estabelece uma chave secreta que pode ser usada em comunicação criptografada para transmissão de dados em uma rede pública de forma segura.

A implementação original do protocolo utiliza raízes primitivas módulo p, onde p é um número primo e g é uma raiz primitiva de p. Essa propriedade permite que a chave secreta possa representar qualquer valor no intervalo de 1 a $p-1$. O protocolo funciona da seguinte forma:

- Alice e Beto escolhem um p e g respeitando a definição acima.
- Alice escolhe um inteiro secreto $a \leq p$ e envia para Beto $A = g^a \pmod p$.
- Beto escolhe um inteiro secreto $b \leq p$ e envia para Alice $B = g^b \pmod p$.
- Alice computa o segredo s com $s = B^a \pmod p$.
- Beto computa o mesmo segredo s com $s = A^b \pmod p$.

O algoritmo explora as propriedades da aritmética modular para que, mesmo quando Alice e Beto não sabem o inteiro secreto do parceiro da comunicação, os dois possam chegar no mesmo segredo s garantidamente pela relação de equivalência:

$$A^b \pmod p \equiv g^{ab} \pmod p \equiv g^{ba} \pmod p \equiv B^a \pmod p$$

Ou de uma forma mais específica e sem variáveis intermediárias:

$$(g^a \pmod p)^b \pmod p \equiv \left(g^b \pmod p\right)^a \pmod p$$

Neste caso, desde que um atacante não descubra os segredos a e b, ele não consegue descobrir a chave secreta. A segurança do protocolo está na inexistência de um algoritmo eficiente para descobrir a, b e $g^{ab} \pmod p$ a partir de $g^a \pmod p$ ou $g^b \pmod p$.

2.1. Exemplo

Utilizando, por exemplo $p = 1987$ e $g = 1484$, que podem ser verificados como primo e raiz primitiva módulo p com os algoritmos do trabalho anterior. Alice e Beto escolhem $a = 1201$ e $b = 1690$, dois números naturais quaisquer e menores que p.

- Alice calcula $A = 1484^{1201} \pmod{1987} = 1842$ e envia para Beto.
- Beto calcula $B = 1484^{1690} \pmod{1987} = 1302$ e envia para Alice.
- Alice calcula $s = 1302^{1201} \pmod{1987} = 1135$.
- Beto calcula $s = 1842^{1690} \pmod{1987} = 1135$.

No final das operações, Alice e Beto podem usar a chave secreta $s = 1135$ para proteger sua transmissão de informação.

Utilizar inteiros muito grandes é uma tarefa complexa, pelo fato de que números maiores que 32 bits precisam ser convertidos para inteiros de precisão arbitrária (*long* em Python), que são lentos para realizar mesmo operações simples, comparando com inteiros puros.

Utilizando ferramentas de *profiling*, é possível verificar que para um número de 24 bits são necessárias centenas de milhares de operações de exponenciação modular (o método `pow`), e em média é testada a primalidade de dezenas de milhares números, o que leva alguns segundos.

Esse tempo cresce de forma exponencial e para 25 bits são necessários minutos, além de que certas sequências de números aleatórios são muito mais lentas. O tempo pode ser muito reduzido utilizando um cache nas funções `phi` e `prime` que são muito utilizadas com os mesmos argumentos:

```
@functools.lru_cache(maxsize=None)
def phi(n):
    # passo base
    if n < 2:
        return 0

    # se n for primo, sera relativamente primo a todos antes dele
    if prime(n):
        return n - 1

    # igual ao if abaixo, apenas evitando testar para os pares tambem
    if (n & 1) == 0:
        m = n >> 1
        return phi(m) << 1 if not (m & 1) else phi(m)

    # testar para todos os primos menores que n, como p nao eh primo
    # havera algum fator entre 3 e n que divide n
    for p in range(3, n, 2):
        if prime(p):
            if n % p:
                continue

            # phi eh multiplicativo, se p divide n (p * o = n) eh
            # possivel utilizar p (menor que n) para calcular phi de n
            o = n // m
            d, partial = math.gcd(m, o), phi(m) * phi(o)
            return partial if d == 1 else partial * d / phi(d)
```

```
@functools.lru_cache(maxsize=None)
def prime(p, tests=20):
    # min evita testar muitas vezes para p pequeno
    for _ in range(min(p, tests)):
        if pow(random.randrange(2, p), p - 1, p) != 1:
            return False
    return True
```

O método `randrange` retorna um inteiro no intervalo $[2, p)$, ou seja, menor que p e maior que 1, utilizando o algoritmo de *Mersenne twister*. Não foi utilizado o algoritmo desenvolvido no trabalho 1 (o *linear congruential generator*) por causa da incapacidade deste de gerar grandes números e pelo *Mersenne twister* já estar embutido de forma otimizada na linguagem.

O código que executa o acordo de chaves, já omitindo as funções acima apresentadas e apresentadas nos trabalhos anteriores, é bastante simples. O código completo se encontra no arquivo `dhke.py` anexo ao trabalho.

```
# digitos desejados no numero primo
d = 10

# classe que representa um lado da troca de chaves
class Partner:
    def __init__(self, p, g):
        self.p, self.g = p, g
        self.secret = random.randrange(1, p)

    # executa o primeiro passo: A = g^a mod p para Alice
    def first_step(self):
        return pow(self.g, self.secret, self.p)

    # executa o segundo passo: s = B^a mod p para Alice
    def second_step(self, other):
        self.s = pow(other, self.secret, self.p)

# tenta encontrar um primo p de d digitos e sua raiz primitiva g
p = None
while not prime(p):
    p = random.randrange(10 ** (d - 1), 10 ** d)
g = primitive_root(p)

# imprime na tela o primo e sua raiz primitiva, dados da chave publica
print('p = {}, g = {}'.format(p, g))

# cria cada um dos participantes da troca com p e g acima definidos
alice = Partner(p, g)
bob = Partner(p, g)

# calcula o segredo (segundo passo) de acordo com o primeiro passo
bob.second_step(alice.first_step())
alice.second_step(bob.first_step())

# imprime o valor final do segredo para fins de comparacao, deve sempre
# imprimir o mesmo numero para A e B
print('As = {}, Bs = {}'.format(alice.s, bob.s))
```

A dificuldade de gerar inteiros de grande tamanho e testar sua primalidade em tempo viável impossibilita a implementação desse protocolo em Python. Uma possibilidade é de implementar em uma linguagem mais rápida como C++ e criar uma interface com Python para utilizar as funções. Outra possibilidade não tão eficiente seria de utilizar múltiplas threads, mas o fator de redução do problema não seria tão grande e haveria o *overhead* de criação das threads.

3. Segurança do algoritmo

3.1. Ataque de *man in the middle*

Como o algoritmo não possui qualquer forma de autenticação, é possível que um atacante monitore a troca de mensagens da negociação de chaves entre Alice e Beto e simule a troca de mensagens entre eles, mas utilizando seu próprio segredo para manipular as mensagens e obter seu conteúdo.

Essa falha pode ser contornada utilizando um canal seguro já estabelecido para troca das mensagens na negociação, sendo então suscetível apenas se o canal seguro for suscetível ao ataque também.

3.2. g raiz primitiva módulo p

Como constatado, existe a necessidade de que g seja raiz primitiva de p para que a troca de chaves funcione corretamente. Na situação em que g não é, o conjunto gerado por g (isto é, os valores de $g^x \pmod p$) para $0 < x < p$) não é o maior conjunto possível (de tamanho $p - 1$), o que seria ideal para o funcionamento do algoritmo.

Para contornar este problema, o número primo p escolhido deve ser suficientemente grande, de forma que na situação em que g não é sua raiz primitiva, o conjunto gerado por g ainda seja suficientemente grande para que não interfira na segurança do algoritmo.