МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теоретических основ компьютерной безопасности и криптографии

**Протоколы открытого распределения ключей**

ОТЧЁТ

ПО ДИСЦИПЛИНЕ

«КРИПТОГРАФИЧЕСКИЕ ПРОТОКОЛЫ»

студента 5 курса 531 группы

специальности 10.05.01 Компьютерная безопасность

факультета компьютерных наук и информационных технологий

Серебрякова Алексея Владимировича

|  |  |  |
| --- | --- | --- |
| Преподаватель  аспирант | \_\_\_\_\_\_\_\_\_\_ | Р. А. Фарахутдинов |
|  | подпись, дата |  |

Саратов 2023

**1 Постановка задачи**

Цель работы:

* Изучение протокола открытого распределения ключей Диффи-Хеллмана и его программная реализация.

Задачи работы:

* Изучить протокол Диффи-Хеллмана, его сильные и слабые стороны;
* Привести программную реализацию протокола.

**2 Теоретические сведения**

Протокол Ди́ффи — Хе́ллмана (англ. Diffie–Hellman key exchange protocol, DH) — криптографический протокол, позволяющий двум и более сторонам получить общий секретный ключ, используя незащищенный от прослушивания канал связи. Полученный ключ используется для шифрования дальнейшего обмена с помощью алгоритмов симметричного шифрования.

Схема открытого распределения ключей, предложенная Диффи и Хеллманом, произвела настоящую революцию в мире шифрования, так как снимала основную проблему классической криптографии — проблему распределения ключей.

В чистом виде алгоритм Диффи — Хеллмана уязвим для модификации данных в канале связи, в том числе для атаки «Man-in-the-middle (человек посередине)», поэтому схемы с его использованием применяют дополнительные методы односторонней или двусторонней аутентификации.

Работу алгоритма можно описать следующим образом. Предположим, существует два абонента: Алиса и Боб. Обоим абонентам известны некоторые два числа и , которые не являются секретными и могут быть известны также другим заинтересованным лицам. Для того, чтобы создать неизвестный более никому секретный ключ, оба абонента генерируют большие случайные числа: Алиса — число , Боб — число . Затем Алиса вычисляет остаток от деления:

и пересылает его Бобу, а Боб вычисляет остаток от деления

и передаёт Алисе. Предполагается, что злоумышленник может получить оба этих значения, но не модифицировать их (то есть, у него нет возможности вмешаться в процесс передачи).

На втором этапе Алиса на основе имеющегося у неё и полученного по сети вычисляет значение:

Боб на основе имеющегося у него и полученного по сети вычисляет значение

Как нетрудно видеть, у Алисы и Боба получилось одно и то же число:

Его они и могут использовать в качестве секретного ключа, поскольку здесь злоумышленник встретится с практически неразрешимой (за разумное время) проблемой вычисления ключа по перехваченным и , если числа выбраны достаточно большими. Работа алгоритма показана на Рисунке 1.

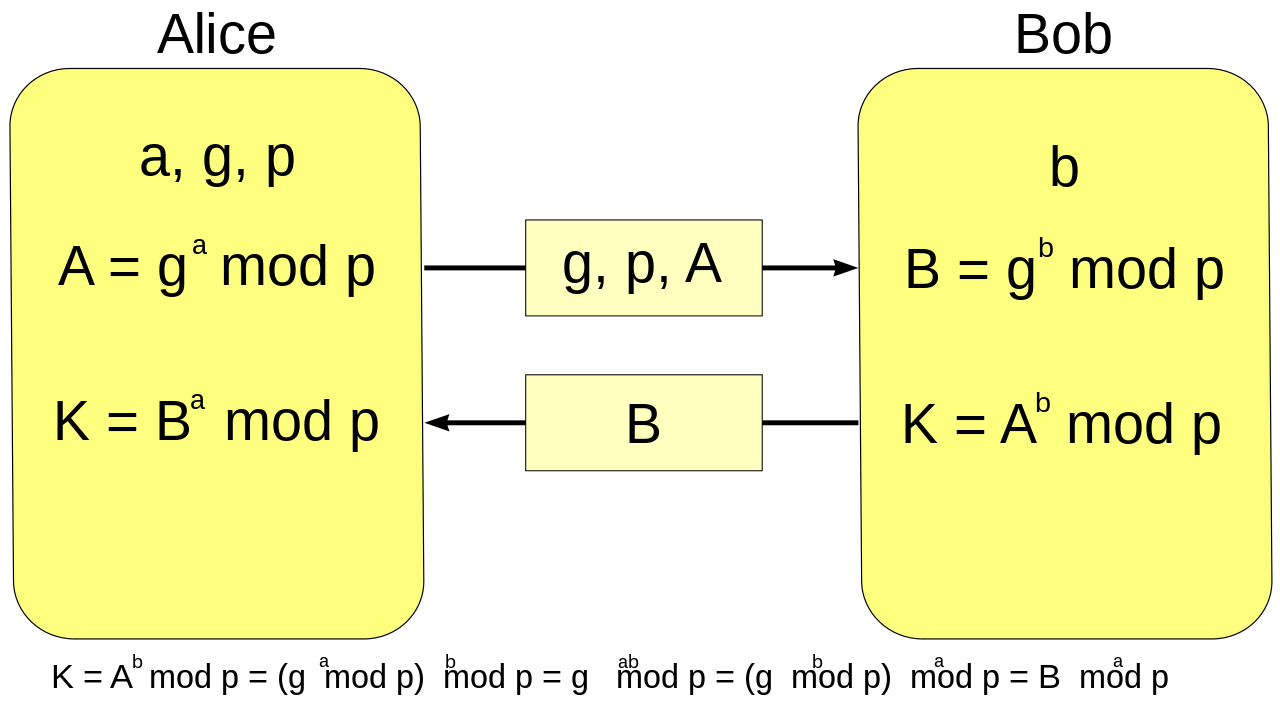


Рисунок 1 - Схема работы алгоритма Диффи-Хеллмана

Опишем алгоритм Диффи-Хеллмана формально.

Общие параметры: – большое простое число, – примитивный корень по модулю .

Протокол:

1. , где – случайное секретное целое число Алисы из интервала .
2. , где – случайное секретное целое число Боба из интервала .
3. (Алиса и Боб не зависимо друг от друга вычисляют их сеансовый ключ ).

**3 Тестирование программы**

На рисунках 2-3 представлены результаты работы программы, эмулирующие оба прохода протокола Диффи-Хеллмана между двумя абонентами сети. В ходе работы программы пользователь вводит режим работы программы ( – с отображением промежуточных шагов, – без отображения промежуточных шагов), а также битовую длину случайно генерируемых простых чисел.

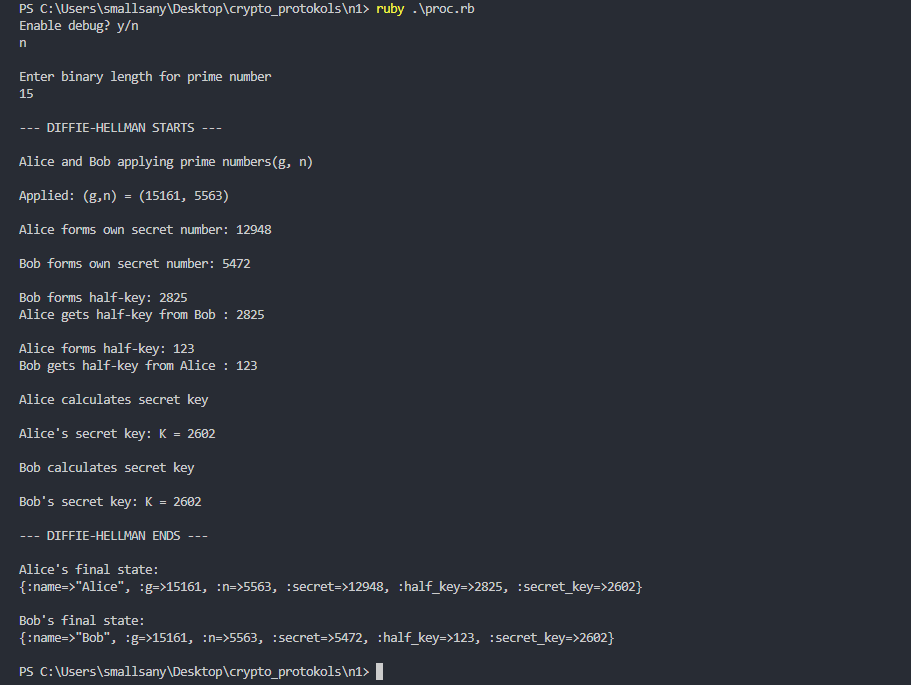


Рисунок 2 - Работа программы

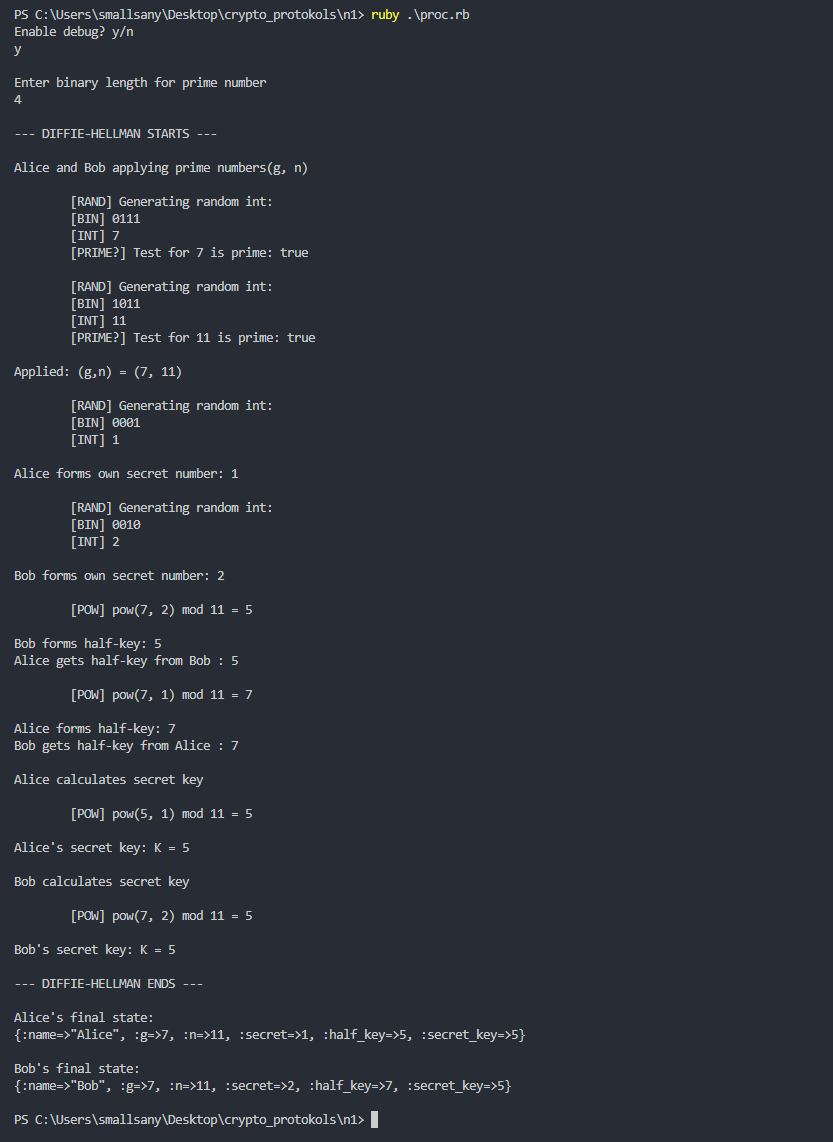


Рисунок 3 - Работа программы (с отображением промежуточных действий)

**ПРИЛОЖЕНИЕ А**

**Код программы proc.rb**

require './methods.rb'

@units = false

if @units

@len = 15

@debug\_mode = false

else

puts %{Enable debug? y/n}

@debug\_mode = gets.strip == 'y'

puts %{\nEnter binary length for prime number}

@len = gets.strip.to\_i

end

@methods = Methods.new({debug\_mode: @debug\_mode})

def get\_prime

first\_num = 0

while ! @methods.is\_prime(first\_num)

first\_num = @methods.gen\_random\_int(@len)

end

first\_num

end

def get\_int

@methods.gen\_random\_int(@len)

end

def apply\_gn

puts %{\nAlice and Bob applying prime numbers(g, n)} if !@units

open\_keys = {g: @methods.find\_primitive\_root(get\_prime), n: get\_prime}

puts %{\nApplied: (g,n) = (#{open\_keys[:g]}, #{open\_keys[:n]})} if !@units

open\_keys

end

def get\_secret\_number(client)

own\_secret = get\_int

puts %{\n#{client[:name]} forms own secret number: #{own\_secret}} if !@units

own\_secret

end

def get\_half\_key(source, destination)

half\_key = @methods.powm(source[:g], source[:secret], source[:n])

puts %{\n#{source[:name]} forms half-key: #{half\_key}} if !@units

puts %{#{destination[:name]} gets half-key from #{source[:name]} : #{half\_key}} if !@units

half\_key

end

def enum\_secret\_key(client)

puts %{\n#{client[:name]} calculates secret key} if !@units

secret\_key = @methods.powm(client[:half\_key], client[:secret], client[:n])

puts %{\n#{client[:name]}'s secret key: K = #{secret\_key}} if !@units

secret\_key

end

def show\_client(client)

puts %{\n#{client[:name]}'s final state:} if !@units

pp client

end

def diffie\_hellman

puts %{\n--- DIFFIE-HELLMAN STARTS ---\n} if !@units

open\_keys = apply\_gn

a\_client = {name: 'Alice', g: open\_keys[:g], n: open\_keys[:n]}

b\_client = {name: 'Bob', g: open\_keys[:g], n: open\_keys[:n]}

a\_client[:secret] = get\_secret\_number(a\_client)

b\_client[:secret] = get\_secret\_number(b\_client)

a\_client[:half\_key] = get\_half\_key(b\_client, a\_client)

b\_client[:half\_key] = get\_half\_key(a\_client, b\_client)

a\_client[:secret\_key] = enum\_secret\_key(a\_client)

b\_client[:secret\_key] = enum\_secret\_key(b\_client)

puts %{\n--- DIFFIE-HELLMAN ENDS ---\n} if !@units

show\_client(a\_client) if !@units

show\_client(b\_client) if !@units

return a\_client[:secret\_key] == b\_client[:secret\_key] if @units

puts

end

diffie\_hellman

**ПРИЛОЖЕНИЕ Б**

**Код программы methods.rb**

class Methods

def initialize(params = {})

@debug\_mode = params.dig(:debug\_mode)

end

def gen\_random\_int(length = 0)

raise 'Not enough number length!' if length == 0

bin\_str = ''

num = 0

while num == 0

length.times do

bin\_str += rand(2).to\_s

end

num = bin\_str.to\_i(2)

end

if @debug\_mode

puts %{\n\t[RAND] Generating random int:}

sleep 0.2

puts %{\t[BIN] #{bin\_str}\n\t[INT] #{num}}

end

num

end

def is\_prime(number = 0)

test = ("1" \* number) !~ /^1?$|^(11+?)\1+$/

puts %{\t[PRIME?] Test for #{number} is prime: #{test}} if @debug\_mode && number != 0

test

end

def powm(key = 0, degree, md)

raise 'Empty key!' if key == 0

result = 1

degree.times do |e|

result = (result \* key) % md

end

puts %{\n\t[POW] pow(#{key}, #{degree}) mod #{md} = #{result}} if @debug\_mode

result

end

def find\_factors(n)

factors = []

(2..Math.sqrt(n).to\_i).each do |i|

while n % i == 0

factors << i

n /= i

end

end

if n > 1

factors << n

end

return factors.uniq

end

def find\_primitive\_root(p)

if !is\_prime(p)

return nil

end

phi = p - 1

factors = find\_factors(phi)

(2...p).each do |g|

is\_primitive\_root = true

factors.each do |factor|

if g.pow(phi / factor, p) == 1

is\_primitive\_root = false

break

end

end

if is\_primitive\_root

return g

end

end

return nil

end

end

**ПРИЛОЖЕНИЕ В**

**Код программы units.rb**

require './proc.rb'

def unit\_test

success\_tries = 0

failure\_tries = 0

total\_tries = 0

puts %{Enter tests count:}

n = gets.strip.to\_i

n.times do

diffie\_hellman

total\_tries += 1

if diffie\_hellman

success\_tries += 1

else

failure\_tries += 0

end

end

puts %{STAT:\n\ttotal: #{total\_tries}\n\tsuccess: #{success\_tries}\n\tfail: #{failure\_tries}}

end

unit\_test