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 g и p, 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 a, Боб — число b. Затем Алиса вычисляет остаток от деления:

$$A = g^a \ mod \ p$$

и пересылает его Бобу, а Боб вычисляет остаток от деления

$$B = g^b \ mod \ p$$

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 a и полученного по сети B вычисляет значение:

$$B^a \ mod \ p = g^{ab} \ mod \ p$$

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

$$A^b \ mod \ p = g^{ab} \ mod \ p$$

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

$$K = g^{ab} \ mod \ p$$

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

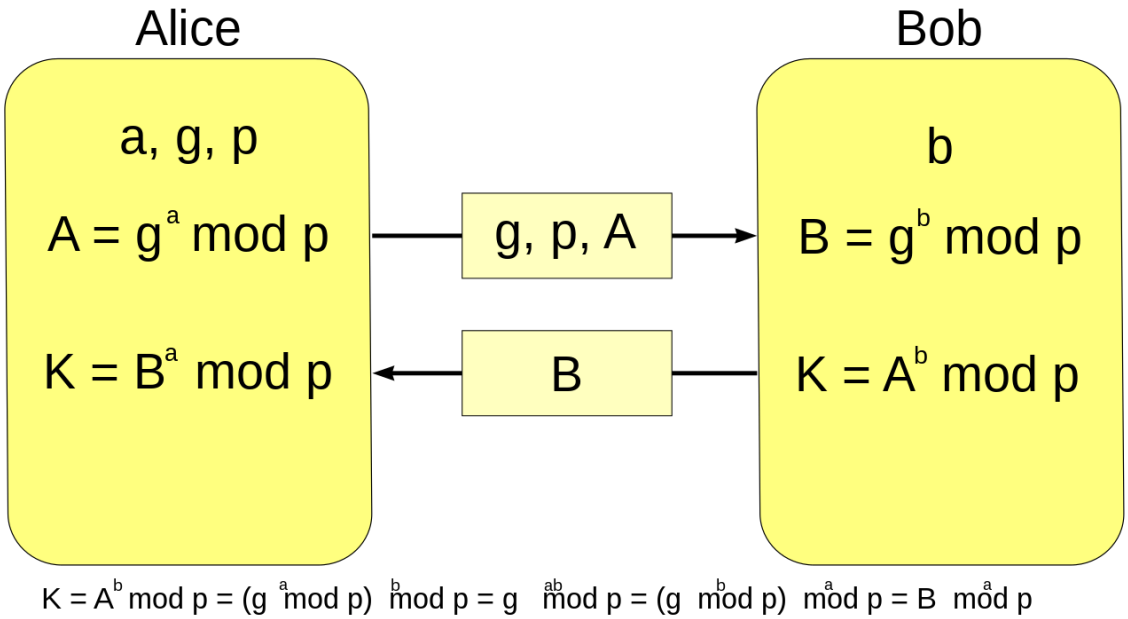

Рисунок 1 - Схема работы алгоритма Диффи-Хеллмана

Опишем алгоритм Диффи-Хеллмана формально.

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

Протокол:

1. $A \rightarrow B: \{X = g^x \ mod \ p\}$, где x – случайное секретное целое число Алисы из интервала $1 < x < p$.
2. $B \rightarrow A: \{Y = g^y \ mod \ p\}$, где y – случайное секретное целое число Боба из интервала $1 < y < p$.

3. $A: K = Y^x \ mod \ p, B: K = X^y \ mod \ p$ (Алиса и Боб не зависимо друг от друга вычисляют их сеансовый ключ K).

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 (y – с отображением промежуточных шагов, n – без отображения промежуточных шагов), а также битовую длину случайно генерируемых простых чисел.

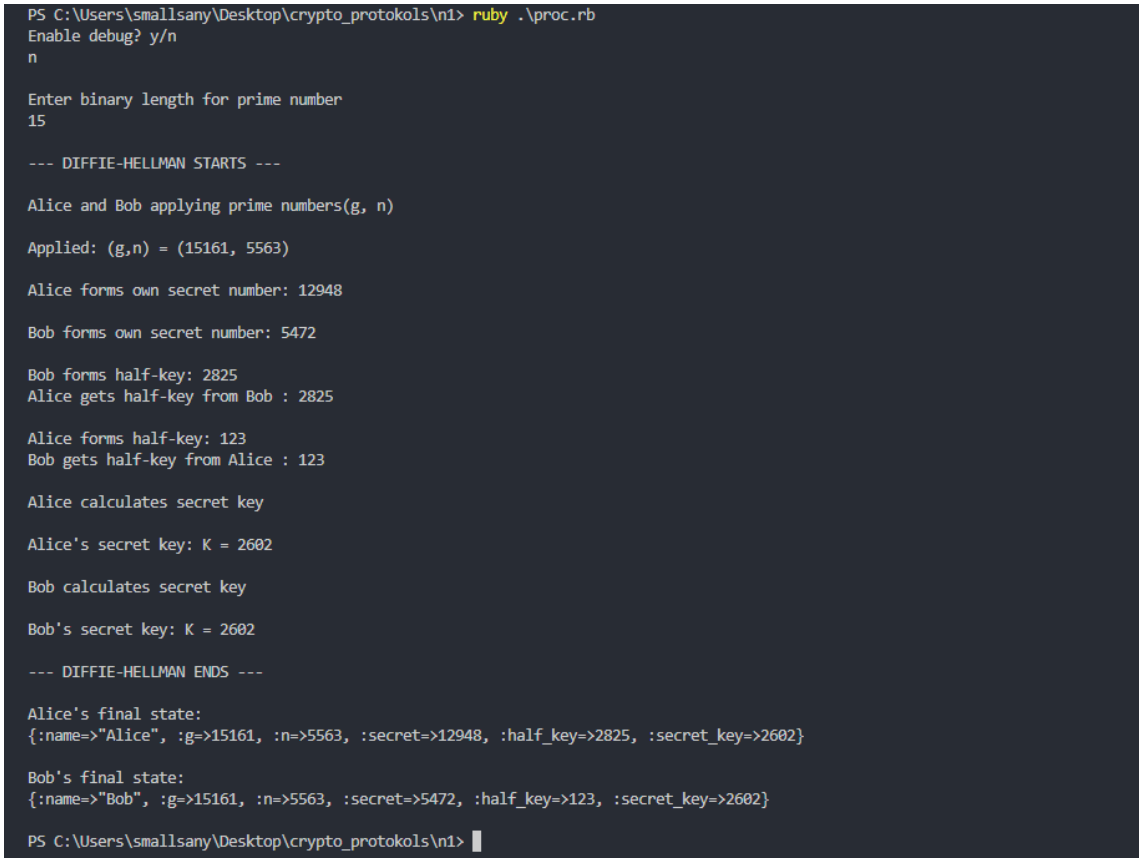

Рисунок 2 - Работа программы

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
```