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 передачи секретного ключа по открытому каналу

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 трехэтапного протокола Шамира передачи секретного ключа по открытому каналу и его программная реализация.

Задачи работы:

- Изучить трехэтапный протокол Шамира, его сильные и слабые стороны;
- Привести программную реализацию протокола.

2 Теоретические сведения

Трёхэтапный протокол Шамира — криптографический трёхэтапный протокол, разработанный Ади Шамиром около 1980 года. Протокол позволяет двум сторонам безопасно обмениваться сообщениями без необходимости распространения ключей шифрования. Обмен сообщением между пользователями происходит в три прохода.

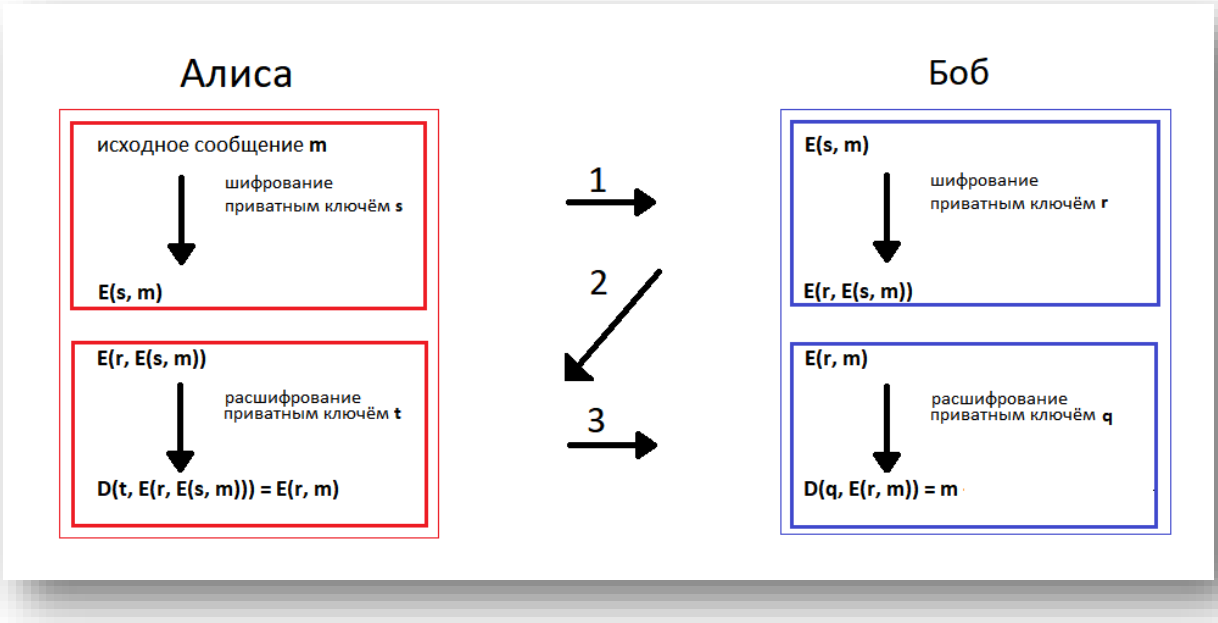

Рисунок 1 - Пример работы трехэтапного протокол Шамира

Используется шифрование на основе функции возведения в степень по модулю. Выбирают достаточно большое простое число $p \sim 2^{1024}$ для которого $p-1$ имеет большой простой множитель. В информационном взаимодействии участвуют два пользователя: Алиса и Боб.

1. Алиса выбирает число a, взаимно простое с $p-1$. Также Алиса использует число a' такое, что $aa'mod(p-1)=1$, то есть $aa'=1+n(p-1), n \in Z$. Алиса шифрует сообщение M и отправляет шифр Бобу:

$$Alice \rightarrow \{E_A(M) = M^a \, mod \, p\} \rightarrow Bob$$

2. Получатель Боб аналогично выбирает целое число b, взаимно простое с $p-1$, и число b' такое, что $bb'mod(p-1)=1$. Боб отправляет обратно следующее сообщение:

$$Bob \rightarrow \{E_B(E_A(M)) = M^{ab} \bmod p\} \rightarrow Alice$$

3. Алиса, получив сообщение, вычисляет $D_A(M^{ab}) = (M^{ab})^{a'} = M^{b(1+n(p-1))} = M^b M^{n(p-1)b} = M^b (M^{p-1})^{nb} = M^b \bmod p$ (используется коммутативность функции возведения в степень по модулю и свойство $M^{p-1} \bmod p = 1$ по малой теореме Ферма) и отправляет Бобу:

$$Alice \rightarrow \{E_B(M) = M^b\} \rightarrow Bob$$

4. Боб расшифровывает сообщение: $D_B(M^b) = (M^b)^{b'} mod\ p = M$.

Если третья сторона перехватила все три сообщения:

$$M_1 = M^a \bmod p$$

$$M_2 = M^{ab} \bmod p$$

$$M_3 = M^b \bmod p$$

Чтобы вычислить M при корректно выбранных параметрах a и b, нужно решить систему из этих трех уравнений, что имеет очень большую вычислительную сложность, так как нужно решать задачу дискретного логарифма.

3 Тестирование программы

На рисунках 2-3 представлены результаты работы программы, эмулирующей работу трехэтапного протокола Шамира между двумя субъектами обмена (Alice и Bob).

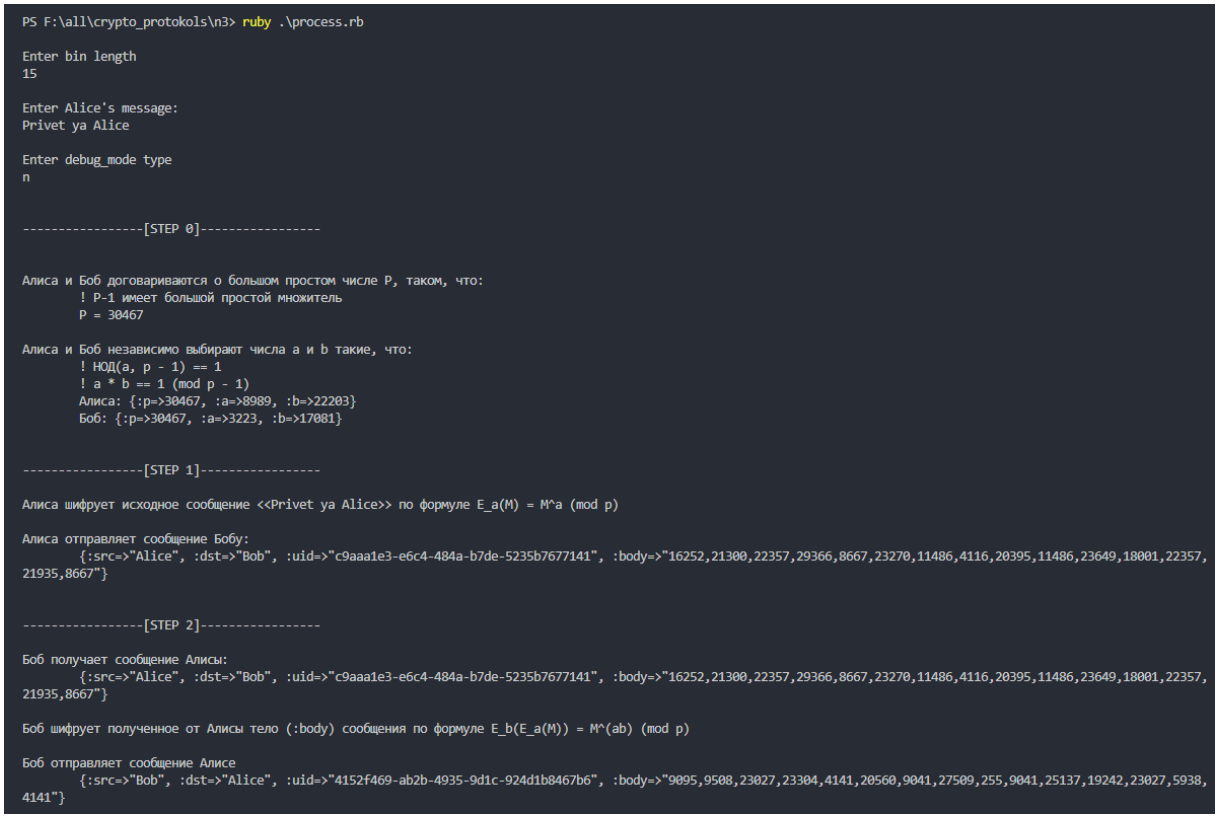

Рисунок 2 - Пример работы программы

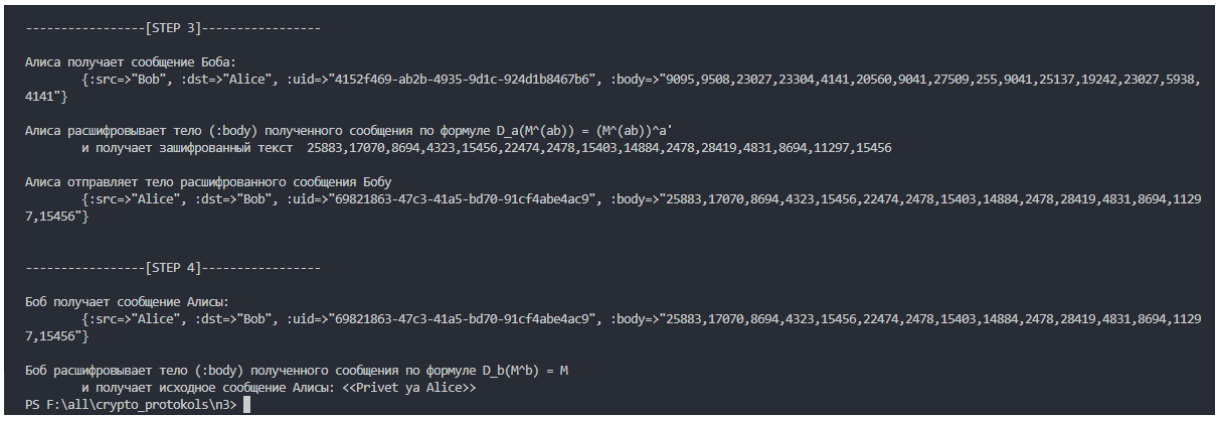

Рисунок 3 - Пример работы программы

ПРИЛОЖЕНИЕ А
Код программы coder.rb

```ruby
class Coder
  def initialize(params = {})
    @p = params.dig(:p).to_i
    @bit_length = params.dig(:bit_length).to_i

    @a, @b = generate_ab(@p)
  end

  def encode(message, type = :raw)
    case type
    when :raw
      encoded_message = message.bytes.map { |c| c.pow(@a, @p) }
      encoded_message.join(",")
    when :enc
      encoded_message = message.split(',').map { |c| c.to_i.pow(@a, @p) }
      encoded_message.join(",")
    end
  end

  def decode(encoded_message, type = :raw)
    case type
    when :raw
      encoded_message.split(",").map { |c| c.to_i.pow(@b, @p) }.pack("C*")
    when :enc
      decoded_message = encoded_message.split(",").map { |c| c.to_i.pow(@b, @p) }
      decoded_message.join(",")
    end
  end

  def get_params
    {
      p: @p,
      a: @a,
      b: @b
    }
  end

  private

  def generate_ab(p)
    a = find_coprime(p - 1)
    b = modular_inverse(a, p - 1)
    [a, b]
  end

  def find_coprime(n)
    random = Random.new
    candidate = random.rand(2..n-1)
```

```ruby
    until candidate.gcd(n) == 1
      candidate = (candidate + 1) % n
    end

    candidate
  end

  def modular_inverse(a, p)
    x, y = extended_gcd(a, p)
    x += p if x < 0
    x
  end

  def extended_gcd(a, b)
    return [0, 1] if a % b == 0
    x, y = extended_gcd(b, a % b)
    [y, x - (a / b) * y]
  end

end
```

ПРИЛОЖЕНИЕ Б
Код программы shamir.rb

```ruby
require './coder.rb'
require 'prime'
require 'securerandom'
require 'openssl'

class Shamir
  def initialize(params = {})
    @debug_mode = params.dig(:debug_mode).to_sym
    @bit_length = params.dig(:bit_length).to_i
    @message = params.dig(:message)
    @int_gen_type = params.dig(:int_gen_type)
  end

  def start
  end

  def step0
    puts "\n\n-----------------[STEP 0]-----------------\n\n"

    prime = gen_large_p(@int_gen_type)

    @alice = make_client('Alice', prime)
    @bob = make_client('Bob', prime)

    puts "\nАлиса и Боб договариваются о большом простом числе P, таком, что:"
    puts "\t! P-1 имеет большой простой множитель\n\tP = #{prime}"
    puts "\nАлиса и Боб независимо выбирают числа a и b такие, что:"
    puts "\t! НОД(a, p - 1) == 1\n\t! a * b == 1 (mod p - 1)"
    puts "\tАлиса: #{@alice[:coder].get_params}\n\tБоб:
#{@bob[:coder].get_params}"


    if @debug_mode == :all
      puts "\nСостояние Алисы"
      pp @alice
      puts "\nСостояние Боба"
      pp @bob
    end

    gets if @debug_mode == :all || @debug_mode == :by_step

  end

  def step1
    puts "\n\n-----------------[STEP 1]-----------------\n\n"
```

```
    send_message(@alice, @bob, @alice[:coder].encode(@message))

    puts "Алиса шифрует исходное сообщение <<#{@message}>> по формуле E_a(M) =
M^a (mod p)"
    puts "\nАлиса отправляет сообщение Бобу:\n\t#{@alice[:m_pushed][-1]}"

    if @debug_mode == :all
      puts "\nСостояние Алисы"
      pp @alice
      puts "\nСостояние Боба"
      pp @bob
    end

    gets if @debug_mode == :all || @debug_mode == :by_step

  end

  def step2
    puts "\n\n-----------------[STEP 2]-----------------\n\n"

    last_msg = @bob[:m_pulled][-1][:body]
    # puts last_msg
    send_message(@bob, @alice, @bob[:coder].encode(last_msg, :enc))

    puts "Боб получает сообщение Алисы:\n\t#{@bob[:m_pulled][-1]}"
    puts "\nБоб шифрует полученное от Алисы тело (:body) сообщения по формуле
E_b(E_a(M)) = M^(ab) (mod p)"
    puts "\nБоб отправляет сообщение Алисе\n\t#{@bob[:m_pushed][-1]}"

    if @debug_mode == :all
      puts "\nСостояние Алисы"
      pp @alice
      puts "\nСостояние Боба"
      pp @bob
    end

    gets if @debug_mode == :all || @debug_mode == :by_step

  end

  def step3
    puts "\n\n-----------------[STEP 3]-----------------\n\n"

    @alice[:m_decoded] << process_mesage(@alice, :enc)
    send_message(@alice, @bob, @alice[:m_decoded][-1][:body])

    puts "Алиса получает сообщение Боба:\n\t#{@alice[:m_pulled][-1]}"
    puts "\nАлиса расшифровывает тело (:body) полученного сообщения по формуле
D_a(M^(ab)) = (M^(ab))^a'"
    puts "\ти получает зашифрованный текст\t#{@alice[:m_decoded][-1][:body]}"
```

```
    puts "\nАлиса отправляет тело расшифрованного сообщения
Бобу\n\t#{@alice[:m_pushed][-1]}"

    if @debug_mode == :all
      puts "\nСостояние Алисы"
      pp @alice
      puts "\nСостояние Боба"
      pp @bob
    end

    gets if @debug_mode == :all || @debug_mode == :by_step

  end

  def step4
    puts "\n\n-----------------[STEP 4]-----------------\n\n"

    @bob[:m_decoded] << process_mesage(@bob, :raw)

    puts "Боб получает сообщение Алисы:\n\t#{@bob[:m_pulled][-1]}"
    puts "\nБоб расшифровывает тело (:body) полученного сообщения по формуле
D_b(M^b) = M"
    puts "\ti получает исходное сообщение Алисы: <<#{@bob[:m_decoded][-
1][:body]}>>"

    if @debug_mode == :all
      puts "\nСостояние Алисы"
      pp @alice
      puts "\nСостояние Боба"
      pp @bob
    end

    gets if @debug_mode == :all || @debug_mode == :by_step

    if @debug_mode == :by_step
      puts "\nСостояние Алисы"
      pp @alice
      puts "\nСостояние Боба"
      pp @bob
    end
  end

  private

  def make_client(name = '', p)
    {
      p: p,
      name: name,
      coder: Coder.new({p: p, bit_length: @bit_length}),
      m_pushed: [],
      m_pulled: [],
```

```
      m_decoded: []
    }
  end

  def send_message(src, dst, m)
    message = {
      src: src[:name],
      dst: dst[:name],
      uid: SecureRandom.uuid,
      body: m
    }

    src[:m_pushed] << message
    dst[:m_pulled] << message
  end

  def process_mesage(client, type = :raw)
    last_message = client[:m_pulled][-1]

    {
      src: last_message[:src],
      dst: last_message[:dst],
      uid: last_message[:uid],
      body: client[:coder].decode(last_message[:body], type)
    }
  end

  def gen_large_p(mode = :default)
    case mode
    when :default
      loop do
        candidate = rand(2 ** @bit_length)
        # pp candidate
        next if candidate.even?

        if Prime.prime?(candidate)
          factors = Prime.prime_division(candidate - 1)
          largest_prime_factor = factors[-1][0]
          return candidate if largest_prime_factor > Math.sqrt(candidate).to_i
        end
      end
    when :alt
      OpenSSL::BN.generate_prime(@bit_length).to_i
    end
  end

end
```

ПРИЛОЖЕНИЕ В
Код программы process.rb

```ruby
require './shamir.rb'

puts "\nEnter bin length"
lng = gets.strip.to_i

puts "\nEnter Alice's message:"
message = gets.strip.to_s

puts "\nEnter debug_mode type"
debug_mode = gets.strip.to_s

params = {
  bit_length: lng,
  debug_mode: debug_mode != '' ? debug_mode : 'by_step',
  message: message,
  int_gen_type: :alt
}

shamir = Shamir.new(params)

shamir.step0
shamir.step1
shamir.step2
shamir.step3
shamir.step4
```