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 обмена ключами

ОТЧЁТ
ПО ДИСЦИПЛИНЕ
«КРИПТОГРАФИЧЕСКИЕ ПРОТОКОЛЫ»

студента 5 курса 531 группы
специальности 10.05.01 Компьютерная безопасность
факультета компьютерных наук и информационных технологий
Бородина Артёма Горовича

Преподаватель
аспирант _____ Р. А. Фарахутдинов
подпись, дата

Саратов 2023

1 Постановка задачи

Необходимо реализовать протокол обмена ключами Нидхема-Шрёдера.

2 Теоретические сведения

В этом протоколе, изобретённом Роджером Нидхемом и Майклом Шрёдером используется симметричная криптография и посредник Трент.

Протокол Нидхема-Шрёдера

1. Алиса посылает Тренту сообщение, содержащее её имя, имя Боба и случайное число: A, B, R_A.
2. Трент генерирует случайный сеансовый ключ. Он шифрует сообщение, содержащее случайный сеансовый ключ и имя Алисы, секретным ключом, общим для него и Боба. Затем он шифрует случайное число Алисы, имя Боба, ключ, и шифрованное сообщение секретным ключом, общим для него и Алисы. Наконец, он отправляет шифрованное сообщение Алисе: $E_A(R_A, B, K, E_B(K, A))$.
3. Алиса расшифровывает сообщение и извлекает K. Она убеждается, что R_A совпадает со значением, отправленным Тренту на этапе (1). Затем она посылает Бобу сообщение, зашифрованное Трентом ключом Боба: $E_B(K, A)$.
4. Боб расшифровывает сообщение и извлекает K. Затем он генерирует другое случайное число, R_B. Он шифрует это число ключом K и отравляет его Алисе: $E_K(R_B)$.
5. Алиса расшифровывает сообщение с помощью ключа K. Она создаёт число $R_B - 1$ и шифрует это число ключом K. Затем она посылает это сообщение обратно Бобу: $E_K(R_B - 1)$.
6. Боб расшифровывает сообщение с помощью ключа K и проверяет значение $R_B - 1$.

Усложнение с параметрами $R_A, R_B, R_B - 1$ служит для предотвращения вскрытия с повторной передачей. При таком способе вскрытия Мэллори

может записать старые сообщения и впоследствии использовать их при попытке взлома протокола. Присутствие R_A на этапе (2) убеждает Алису, что сообщение Трента достоверно и не является повторной передачей отклика, использованного при одном из предыдущих применений протокола. Когда Алиса успешно расшифрует R_B и передаёт Бобу $R_B - 1$ на этапе (5), Боб убеждается, что сообщения Алисы не являются повторной передачей сообщения, использованных при одном из предыдущих применений протокола.

3 Практическая реализация

3.1 Описание программы

Язык программной реализации – Common Lisp. Выполнение программы разбито на шаги. Используемый алгоритм асимметричного шифрования – RSA. Используемый алгоритм симметричного шифрования – тривиальная операция XOR со значением ключа. По ходу выполнения программа описывает выполняемые действия и отображает получаемые значения.

3.2 Результаты тестирования программы

Рассмотрим процесс выполнения программы со значением битовой длины l модуля RSA, равное 128.

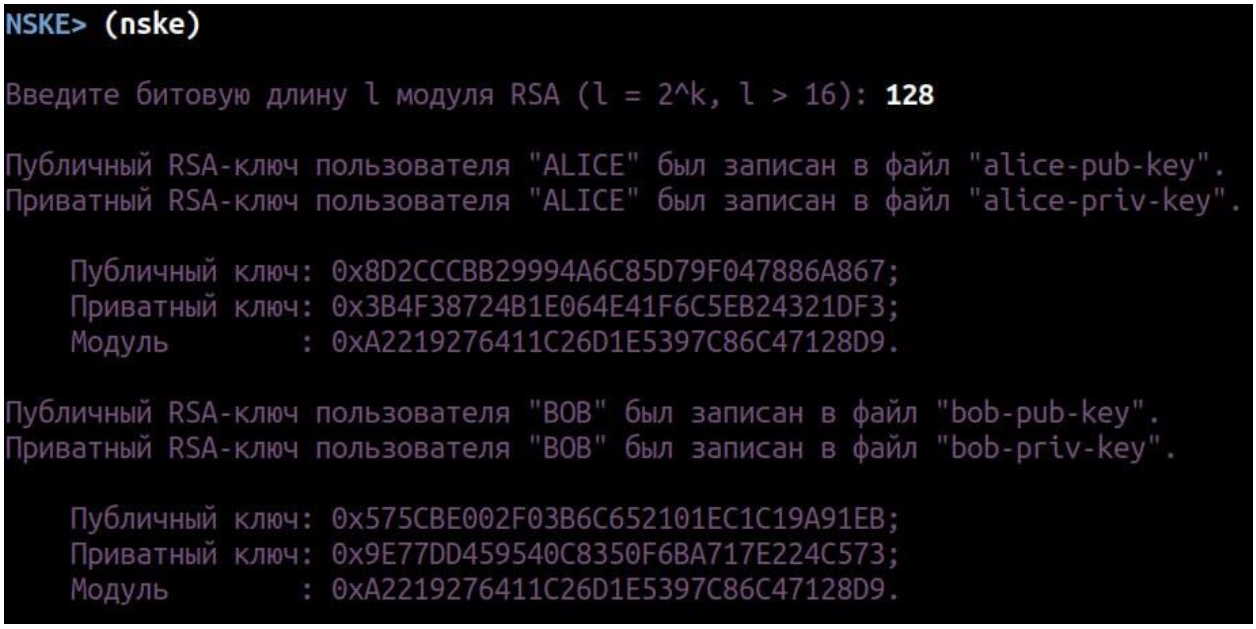

Рисунок 1 – Процесс генерации RSA-ключей для пользователей ALICE и BOB

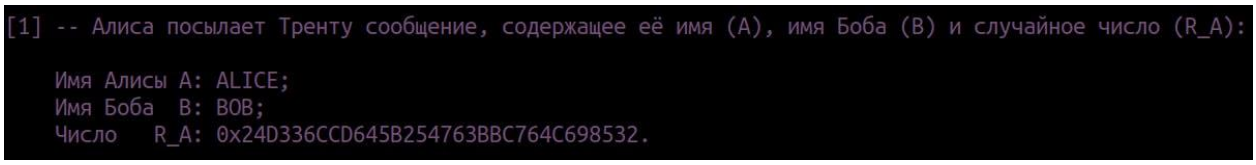

Рисунок 2 – Выполнение шага 1 протокола

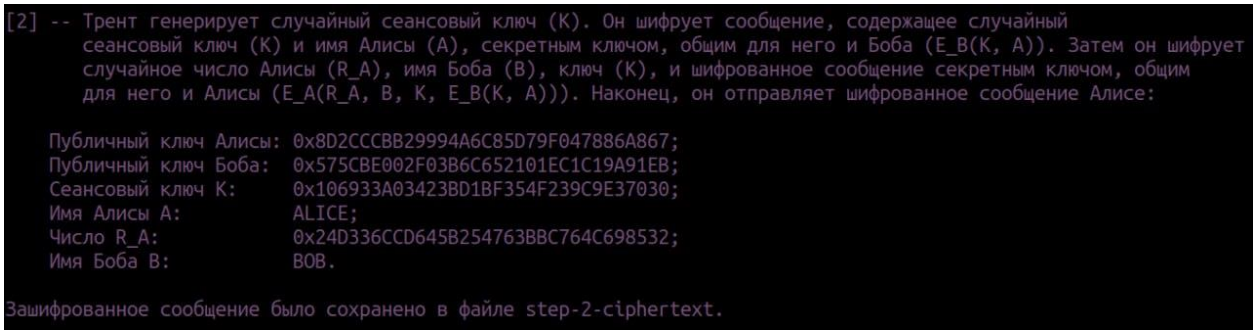

Рисунок 3 – Выполнение шага 2 протокола

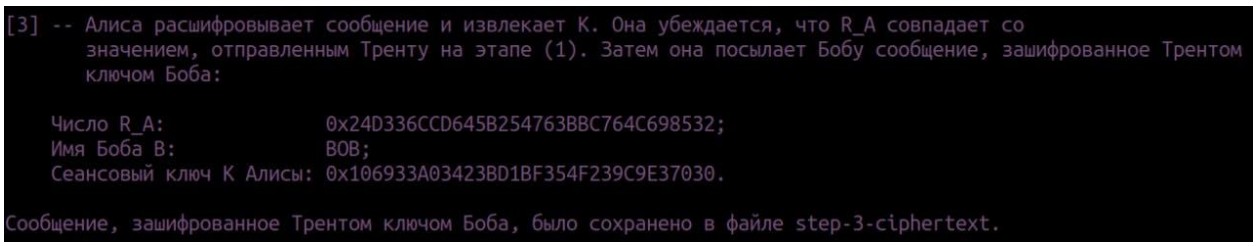

Рисунок 4 – Выполнение шага 3 протокола

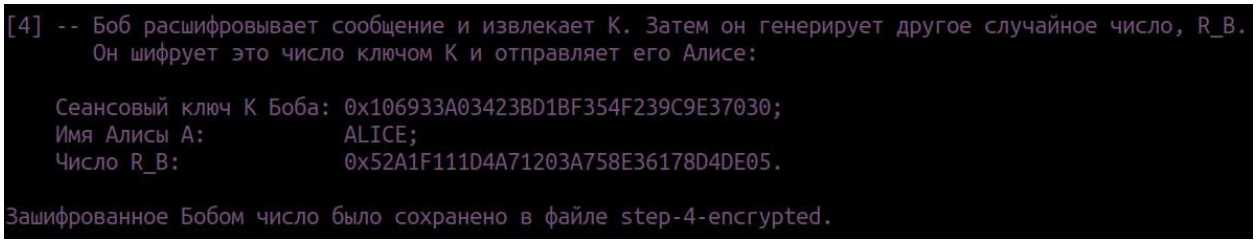

Рисунок 5 – Выполнение шага 4 протокола

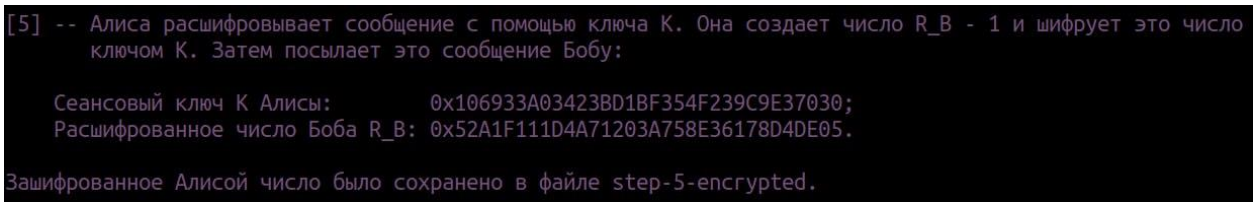

Рисунок 6 – Выполнение шага 5 протокола

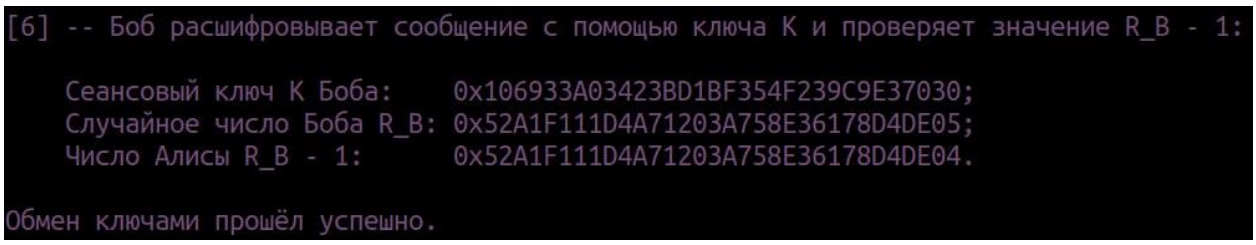

Рисунок 7 – Выполнение шага 6 протокола

Листинг программы

```
(defpackage #:aux
  (:use :cl))

(in-package #:aux)



(defmacro while (condition &body body)
  `(loop while ,condition
         do (progn ,@body)))



(defun write-to-file (data filename)
  (with-open-file (out filename :direction :output :if-exists
:supersede
                                :if-does-not-exist :create)
    (dolist (param data)
      (format out "~a~%" param))))



(defun n-elts (elt n)
  (if (> n 1)
      (list n elt)
      elt))



(defun compr (elt n lst)
  (if (null lst)
      (list (n-elts elt n))
      (let ((next (car lst)))
        (if (eql next elt)
            (compr elt (1+ n) (cdr lst))
            (cons (n-elts elt n) (compr next 1 (cdr lst)))))))
```

```
(defun compress (x)
  (if (consp x)
      (compr (car x) 1 (cdr x))
      x))


(defun is-pow-of-2? (num)
  (zerop (logand num (1- num))))


(defun mod-expt (base power modulo)
  (setq base (mod base modulo))
  (do ((product 1)) ((zerop power) product)
    (do () ((oddp power))
      (setq base (mod (* base base) modulo)
            power (ash power -1)))
    (setq product (mod (* product base) modulo)
          power (1- power))))


(defun miller-rabin (n &optional (k 10))
  (when (or (= 2 n) (= 3 n)) (return-from miller-rabin t))
  (when (or (< n 2) (= 0 (logand n 1))) (return-from miller-rabin))
  (let* ((n-pred (1- n)) (bound (- n-pred 2)) (t-val n-pred) (s 0)
(round 0) (x))
    (while (= 0 (logand t-val 1)) (setq s (1+ s) t-val (ash t-val -
1)))
    (do () (nil)
      (tagbody next-iteration
         (when (= k round) (return-from miller-rabin t))
         (setq x (mod-expt (+ 2 (random bound)) t-val n))
         (when (or (= 1 x) (= n-pred x))
           (incf round) (go next-iteration))
```

```
         (do ((iter 0 (1+ iter))) ((= iter (1- s)) (return-from
miller-rabin))
           (setq x (mod (* x x) n))
           (when (= 1 x) (return-from miller-rabin))
           (when (= n-pred x)
             (incf round) (go next-iteration)))))))


(defparameter *base-primes*
  (remove-if-not #'(lambda (prime?) (miller-rabin prime? 12))
                 (loop for prime? from (1+ (ash 1 15)) to (1- (ash 1
16)) by 2
                       collect prime?)))


(defun ext-gcd (a b)
  (let ((s 0) (old-s 1) (r b) (old-r a)
        (quotient) (bezout-t))
    (while (not (zerop r))
      (setq quotient (floor old-r r))
      (psetq old-r r r (- old-r (* quotient r))
             old-s s s (- old-s (* quotient s))))
    (if (zerop b) (setq bezout-t 0)
        (setq bezout-t (floor (- old-r (* old-s a)) b)))
    (list old-r old-s bezout-t)))


(defun rho-pollard-machinerie (n x-0 &optional (c 1) (rounds 1000))
  (when (miller-rabin n) (return-from rho-pollard-machinerie 'PRIME))
  (let ((mapping (lambda (x) (mod (+ c (* x x)) n)))
        (a x-0) (b x-0) (round 0) (q))
    (tagbody map
       (incf round)
```

```
       (when (> round rounds) (return-from rho-pollard-machinerie
'GEN-NEW))
       (setq a (funcall mapping a)
             b (funcall mapping (funcall mapping b))
             q (gcd (- a b) n))
       (cond ((< 1 q n) (return-from rho-pollard-machinerie
                          (list q (miller-rabin q))))
             ((= n q) (return-from rho-pollard-machinerie))
             (t (go map))))))



(defun rho-pollard-wrapper (n x-0)
  (let ((c 1) (head) (factor) (factors))
    (while (zerop (logand n 1))
      (setq factors (cons 2 factors) n (ash n -1)))
    (setq x-0 (mod x-0 n))
    (while (/= 1 n)
      (setq factor (rho-pollard-machinerie n x-0 c))
      (cond ((eql 'PRIME factor) (setq factors (cons n factors) n 1))
            ((eql 'GEN-NEW factor) (return))
            ((cadr factor) (setq factors (cons (setq head (car
factor)) factors)
                                 n (/ n head)))
            ((null factor) (while (= (- n 2) (setq c (1+ (random (1-
n)))))))
            (t (setq n (/ n (setq head (car factor)))
                     factors (append factors
                                     (rho-pollard-wrapper head (random
head)))))))
    factors))



(defun rho-pollard (n x-0)
  (let* ((factors (rho-pollard-wrapper n x-0)))
```

```
    (when (null factors) (return-from rho-pollard))
    (when (= n (apply #'* factors))
      (compress (sort (rho-pollard-wrapper n x-0) #'<)))))


(defun get-bit-len ()
  (let ((bit-len))
    (tagbody try-again
       (setq bit-len (read))
       (when (and (integerp bit-len) (is-pow-of-2? bit-len) (> bit-len
16))
         (return-from get-bit-len bit-len))
       (format t "Некорректный ввод! Попробуйте снова: ")
       (go try-again))))


(defun find-g (p)
  (when (not (miller-rabin p)) (return-from find-g))
  (let ((phi (1- p)) (factors) (g?) (bound (- p 2)))
    (setq factors (rho-pollard phi (random p)))
    (when (null factors) (return-from find-g))
    (setq factors (mapcar #'(lambda (factor) (cond ((atom factor)
factor)
                                                   (t (cadr factor))))
factors)
          factors (mapcar #'(lambda (factor) (floor phi factor))
factors))
    (tagbody try-again
       (setq g? (+ 2 (random bound)))
       (when (= 1 (mod-expt g? (car factors) p)) (go try-again))
       (when (remove-if-not #'(lambda (pow) (= 1 (mod-expt g? pow p)))
                            factors) (go try-again))) g?))
```

```
(defun generate-even (target-len)
  (apply #'+ (ash 1 (1- target-len))
         (mapcar #'(lambda (bit pow) (* bit (ash 1 pow)))
                 (append (loop for bit from 0 to (- target-len 3)
                               collect (random 2)) '(0))
                 (loop for pow from (- target-len 2) downto 0 collect
pow))))


(defun generate-prime (target-len)
  (when (not (is-pow-of-2? target-len))
    (return-from generate-prime))
  (when (= 16 target-len)
    (return-from generate-prime (nth (random (length *base-primes*))
                                     *base-primes*)))
  (let ((prime) (s) (prime?) (req-len (- target-len 16)))
    (tagbody pick-prime
       (setq prime (nth (random (length *base-primes*)) *base-
primes*))
       (when (not (miller-rabin prime)) (go pick-prime)))
    (tagbody try-again
       (setq s (generate-even req-len)
             prime? (1+ (* prime s)))
       (if (and (= 1 (mod-expt 2 (1- prime?) prime?))
                (/= 1 (mod-expt 2 s prime?))
                (zerop (logxor (length (write-to-string prime? :base
2))
                               target-len)))
           (return-from generate-prime prime?)
           (go try-again)))))


(defun gen-p&g (bit-len)
  (let ((p) (g))
```

```
(tagbody gen-prime
   (setq p (generate-prime bit-len)
         g (find-g p))
   (when (null g) (go gen-prime)))
(list p g)))
```

```
(defpackage #:rsa
  (:use :cl))

(in-package #:rsa)


; (defparameter e 65537)


(defun rsa-machinerie (key-length &optional (num-users 1))
  (when (not (aux::is-pow-of-2? key-length))
    (return-from rsa-machinerie))
  (let ((p) (q) (n) (phi) (e) (d) (res))
    (tagbody try-again
       (setq p (aux::generate-prime (ash key-length -1))
             q (aux::generate-prime (ash key-length -1))
             n (* p q))
       (when (not (zerop (logxor (length (write-to-string n :base 2))
                                 key-length)))
         (go try-again)))
    (do ((i 0 (1+ i))) ((= num-users i) res)
      (setq phi (* (1- p) (1- q)))
      (aux::while (/= 1 (gcd (setq e (random phi)) phi)))
      (setq d (mod (cadr (aux::ext-gcd e phi)) phi)
            res (cons (list e d n) res)))))


(defun rsa-generate-keys (key-length prefixes)
  (let ((keys (rsa-machinerie key-length (length prefixes)))
        (prefix) (name) (pub-filename) (priv-filename))
    (do ((i 0 (1+ i))) ((= (length prefixes) i))
      (destructuring-bind (e d n) (nth i keys)
        (setq prefix (nth i prefixes)
              name (string-upcase (nth i prefixes))
```

```
              pub-filename (concatenate 'string prefix  "-pub-key")
              priv-filename (concatenate 'string prefix "-priv-key"))
        (aux::write-to-file (list name e n)  pub-filename)
        (format t "Публичный RSA-ключ пользователя ~s был записан в
файл ~s.~%"
                name pub-filename)
        (aux::write-to-file (list name d n) priv-filename)
        (format t "Приватный RSA-ключ пользователя ~s был записан в
файл ~s.~%"
                name priv-filename)
        (format t "~%    Публичный ключ: 0x~x;
    Приватный ключ: 0x~x;
    Модуль        : 0x~x.~%" e d n) (terpri)))))
```

```
(defpackage #:nske
  (:use :cl))

(in-package #:nske)


(defun stop () (read-line))


(defun read-parse (filename &optional (at 0))
  (parse-integer (uiop:read-file-line filename :at at)))


(defun step-1 ()
  (let ((name-alice) (name-bob) (random-num) (bound))
    (format t "~%[1] -- Алиса посылает Тренту сообщение, содержащее её
имя (A), имя Боба (B) и случайное число (R_A): ") (stop)
    (setq name-alice (uiop:read-file-line "alice-pub-key")
          name-bob   (uiop:read-file-line   "bob-pub-key")
          bound      (read-parse "alice-pub-key" 2)
          random-num (random bound))
    (format t "~%    Имя Алисы A: ~a;
    Имя Боба  B: ~a;
    Число   R_A: 0x~x.~%" name-alice name-bob random-num)
    (aux::write-to-file (list name-alice name-bob random-num) "step-1-
message")
    (format t "~%Сообщение было сохранено в файл step-1-message.~%")))


(defun step-2 ()
  (let ((bound) (k) (name-alice) (key-bob) (ciphertext)
        (random-alice) (name-bob) (key-alice))
    (format t "~%[2] -- Трент генерирует случайный сеансовый ключ (K).
Он шифрует сообщение, содержащее случайный
```

```
       сеансовый ключ (K) и имя Алисы (A), секретным ключом, общим для
него и Боба (E_B(K, A)). Затем он шифрует
       случайное число Алисы (R_A), имя Боба (B), ключ (K), и
шифрованное сообщение секретным ключом, общим
       для него и Алисы (E_A(R_A, B, K, E_B(K, A))). Наконец, он
отправляет шифрованное сообщение Алисе: ") (stop)
    (setq bound (read-parse "alice-pub-key" 2) k (random bound)
          name-alice (uiop:read-file-line "step-1-message")
          key-bob (read-parse "bob-pub-key" 1)
          ciphertext (mapcar #'(lambda (num) (aux::mod-expt num key-
bob bound))
                             (list k (mod (sxhash name-alice) bound)))
          random-alice (read-parse "step-1-message" 2)
          name-bob (uiop:read-file-line "step-1-message" :at 1)
          key-alice (read-parse "alice-pub-key" 1)
          ciphertext (mapcar #'(lambda (num) (aux::mod-expt num key-
alice bound))
                             (append (list random-alice (mod (sxhash
name-bob) bound) k)
                                     ciphertext)))
    (format t "~%    Публичный ключ Алисы: 0x~x;
    Публичный ключ Боба:  0x~x;
    Сеансовый ключ K:     0x~x;
    Имя Алисы A:          ~a;
    Число R_A:            0x~x;
    Имя Боба B:           ~a.~%"
            key-alice key-bob k name-alice random-alice name-bob)
    (aux::write-to-file ciphertext "step-2-ciphertext")
    (format t "~%Зашифрованное сообщение было сохранено в файле step-
2-ciphertext.~%")))


(defun step-3 ()
  (let ((bound) (priv-alice) (plaintext) (random-alice)
```

```
        (random-alice-step-1) (hash-bob) (k) (name-bob))
    (format t "~%[3] -- Алиса расшифровывает сообщение и извлекает K. Она убеждается, что R_A совпадает со
       значением, отправленным Тренту на этапе (1). Затем она посылает Бобу сообщение, зашифрованное Трентом
       ключом Боба: ") (stop)
    (setq bound (read-parse "alice-pub-key" 2)
          priv-alice (read-parse "alice-priv-key" 1)
          plaintext (mapcar #'(lambda (num) (aux::mod-expt num priv-alice bound))
                            (mapcar #'parse-integer (uiop:read-file-lines "step-2-ciphertext")))
          random-alice (nth 0 plaintext) hash-bob (nth 1 plaintext)
          random-alice-step-1 (read-parse "step-1-message" 2)
          k (nth 2 plaintext) name-bob (uiop:read-file-line "bob-pub-key"))
    (when (/= random-alice random-alice-step-1)
      (format t "~%Расшифрованное значение R_A не совпадает с отправленным на первом шаге! Экстренное завершение протокола.~%")
      (return-from step-3 t))
    (format t "~%    Число R_A:              0x~x;~%" random-alice)
    (when (/= (mod (sxhash name-bob) bound) hash-bob)
      (format t "~%Расшифрованное имя Боба не совпадает с отправленным на первом шаге! Экстренное завершение протокола.~%")
      (return-from step-3 t))
    (format t "    Имя Боба B:             ~a;
    Сеансовый ключ K Алисы: 0x~x.~%" name-bob k)
    (aux::write-to-file (list k) "sess-alice")
    (aux::write-to-file (list (nth 3 plaintext) (nth 4 plaintext)) "step-3-ciphertext")
    (format t "~%Сообщение, зашифрованное Трентом ключом Боба, было сохранено в файле step-3-ciphertext.~%")))
```

```
(defun step-4 ()
  (let ((bound) (priv-bob) (name-alice) (hash-alice) (plaintext) (k)
(random-bob))
    (format t "~%[4] -- Боб расшифровывает сообщение и извлекает K.
Затем он генерирует другое случайное число, R_B.
       Он шифрует это число ключом K и отправляет его Алисе: ") (stop)
    (setq bound (read-parse "bob-pub-key" 2)
          priv-bob (read-parse "bob-priv-key" 1)
          name-alice (uiop:read-file-line "alice-pub-key")
          plaintext (mapcar #'(lambda (num) (aux::mod-expt num priv-
bob bound))
                            (mapcar #'parse-integer (uiop:read-file-
lines "step-3-ciphertext")))
          k (nth 0 plaintext) hash-alice (nth 1 plaintext)
          random-bob (random bound))
    (format t "~%    Сеансовый ключ K Боба: 0x~x;" k)
    (when (/= (mod (sxhash name-alice) bound) hash-alice)
      (format t "~2%Полученное имя Алисы не совпадает с
действительным! Экстренное завершение протокола.~%")
      (return-from step-4 t))
    (format t "~%    Имя Алисы A:           ~a;
    Число R_B:             0x~x.~%" name-alice random-bob)
    (aux::write-to-file (list k) "sess-bob")
    (aux::write-to-file (list (logxor random-bob k)) "step-4-
encrypted"))
    (format t "~%Зашифрованное Бобом число было сохранено в файле
step-4-encrypted.~%"))


(defun step-5 ()
  (format t "~%[5] -- Алиса расшифровывает сообщение с помощью ключа
K. Она создает число R_B - 1 и шифрует это число
       ключом K. Затем посылает это сообщение Бобу: ") (stop)
  (let ((sess) (encrypted) (decrypted))
```

```
    (setq sess (read-parse "sess-alice")
          encrypted (read-parse "step-4-encrypted")
          decrypted (logxor sess encrypted))
    (format t "~%    Сеансовый ключ K Алисы:        0x~x;
    Расшифрованное число Боба R_B: 0x~x.~%" sess decrypted)
    (aux::write-to-file (list (logxor (1- decrypted) sess)) "step-5-
encrypted")
    (format t "~%Зашифрованное Алисой число было сохранено в файле
step-5-encrypted.~%")))


(defun step-6 ()
  (format t "~%[6] -- Боб расшифровывает сообщение с помощью ключа K и
проверяет значение R_B - 1: ") (stop)
  (let ((sess) (random-bob) (random-bob-pred?))
    (setq sess (read-parse "sess-bob")
          random-bob (logxor (read-parse "step-4-encrypted") sess)
          random-bob-pred? (logxor (read-parse "step-5-encrypted")
sess))
    (format t "~%    Сеансовый ключ K Боба:    0x~x;
    Случайное число Боба R_B: 0x~x;
    Число Алисы R_B - 1:      0x~x.~%" sess random-bob random-bob-
pred?)
    (if (zerop (logxor random-bob (1+ random-bob-pred?)))
        (format t "~%Обмен ключами прошёл успешно.~%")
        (format t "~%Обмен ключами не удался.~%"))))


(defun nske ()
  (let ((bit-len))
    (format t "~%Введите битовую длину l модуля RSA (l = 2^k, l > 16):
")
    (setq bit-len (aux::get-bit-len)) (terpri)
    (rsa::rsa-generate-keys bit-len '("alice" "bob"))
```

```
          (step-1)                          (step-2)
    (when (step-3) (return-from nske)) (when (step-4) (return-from
nske))
          (step-5)                          (step-6)))
```