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
Бородина Артёма Горовича

Преподаватель
аспирант _____ Р. А. Фарахутдинов
подпись, дата

Саратов 2023

1 Постановка задачи

Необходимо реализовать протокол открытого распределения ключей «станция-станция».

2 Теоретические сведения

Обмен ключами Диффи-Хеллмана чувствителен к атаке «человек посередине», так же, как и протокол Хьюза. Одним из способов предотвратить такую атаку является подпись Алисой и Бобом сообщений, которые они посылают друг другу. Применяемый при этом протокол предполагает, что у Алисы есть открытый ключ Боба, а у Боба есть открытый ключ Алисы.

Вот как с помощью протокола «станция-станция» можно провести алгоритм Диффи-Хеллмана, защищаясь от атаки «человек посередине».

Протокол «станция-станция»

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

1. $A \to B: \{X = g^x \ mod \ p\}$, $1 < x < p, x$ – секретное большое число Алисы.
2. 2.1. Боб выбирает случайное секретное большое число y $(1 < y < p)$;
   2.2. Боб вычисляет $Y = g^y \ mod \ p$ и $K = X^y \ mod \ p$;
   2.3. Боб подписывает X и Y, вычисляя подпись $S_B(X, Y)$;
   2.4. Боб шифрует подпись $E_K(S_B(X, Y))$ ключом K;
   2.5. $B \to A: \{Y, E_K(S_B(X, Y))\}$.
3. 3.1. Алиса вычисляет $K = Y^x \ mod \ p$;
   3.2. Алиса расшифровывает $D_K\left(E_K(S_B(X, Y))\right) = S_B(X, Y)$ подпись Боба и проверяет её, если подпись верна, то протокол продолжается;
   3.3. Алиса вычисляет свою подпись $S_A(X, Y)$ и шифрует её $E_K(S_A(X, Y))$;

3.4. $A \rightarrow B: \{E_K(S_A(X, Y))\}$.

Боб расшифровывает и проверяет подпись Алисы, если она верна, то протокол заканчивается положительно.

3 Практическая реализация

3.1 Описание программы

Язык программной реализации – Common Lisp. Программа разбита на шаги. В качестве предварительного этапа выполнения протокола осуществляется генерация открытого ключа проверки подписи и секретного ключа формирования подписи пользователей ALICE и BOB, а также генерация секретного большого простого числа p и первообразного корня по модулю этого числа p. Все шаги программы выполняются в соответствии с протоколом. Подпись была сгенерирована и проверена в соответствии с протоколом цифровой подписи Шнорра на эллиптических кривых, в качестве алгоритма шифрования использовалась тривиальная операция XOR. В качестве вспомогательных использовались функции проверки числа на простоту тестом Миллера-Рабина, быстрое возведение в степень по модулю, расширенный алгоритм Евклида и т.д.

3.2 Результаты тестирования программы

Рассмотрим работу протокола при введённых значениях битовой длины l подписи, равное 128 и битовой длины модуля p, равное 128.

После ввода желаемой битовой длины подписи программа сгенерирует значения открытого ключа проверки подписи и секретного ключа формирования подписи для пользователей ALICE и BOB. Для протокола цифровой подписи Шнорра на эллиптической кривой открытый ключ проверки подписи содержит уравнение эллиптической кривой (в нашем случае значение p простого модуля, над полем характеристики которого определена кривая, а также коэффициент b из уравнения $y^2 = x^3 + ax + b$), образующую

Q простого порядка r и точку $P = lQ$. Секретным ключом формирования подписи является показатель l. Таким образом, файл пользователя его открытого ключа проверки подписи содержит следующие значения в указанном порядке: 1) значение простого модуля p; 2) значение коэффициента b уравнения кривой; 3) значение r простого порядка образующей точки Q; 4) координаты x и y образующей точки Q; 5) координаты точки $P = lQ$.

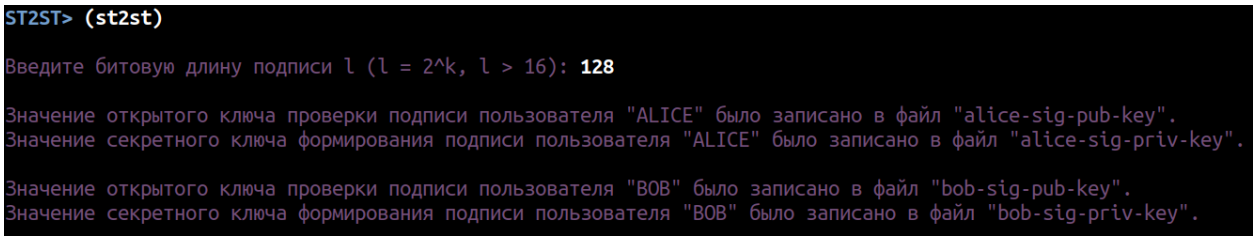

Рисунок 1 – Генерация ключей пользователей

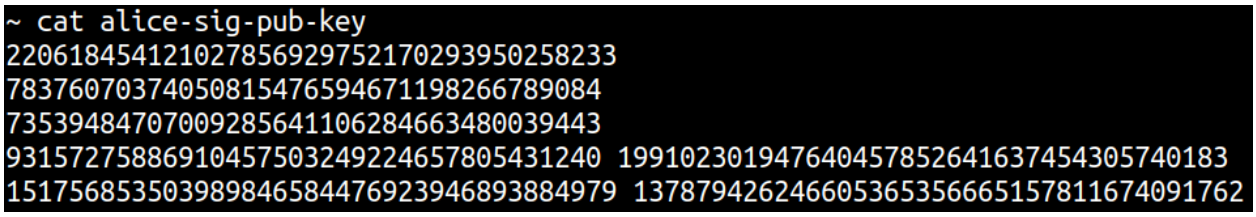

Рисунок 2 – Значение открытого ключа проверки подписи пользователя ALICE

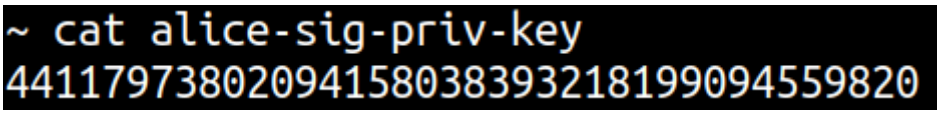

Рисунок 3 – Значение секретного ключа формирования подписи пользователя ALICE

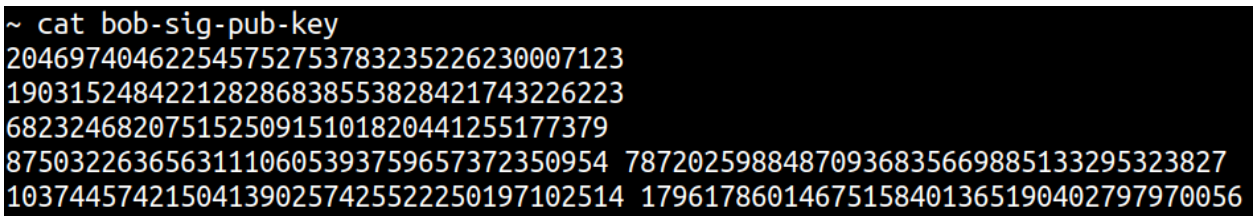

Рисунок 4 – Значение открытого ключа проверки подписи пользователя BOB

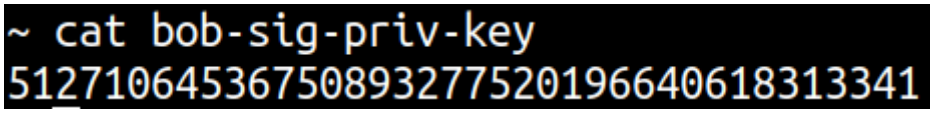

Рисунок 5 – Значение секретного ключа формирования подписи пользователя BOB

Далее программа запросит битовую длину модуля p, которая принимает участие в генерации простого числа, используемого уже по ходу самого протокола «станция-станция». Для этого числа будет сгенерирован первообразный корень, участвующий в дальнейших вычислениях. Для компактности все числа (за исключением помещённых в файл) представлены в шестнадцатеричной системе счисления.

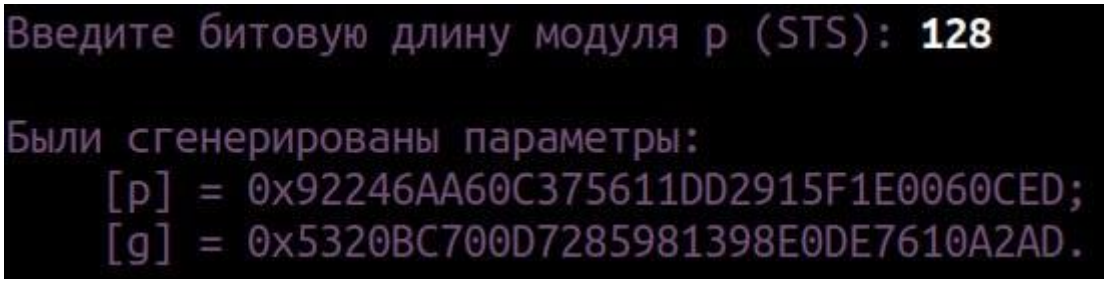

Рисунок 6 – Результат генерации параметров p и g

Далее выполнение программа происходит по шагам, описанным в протоколе.

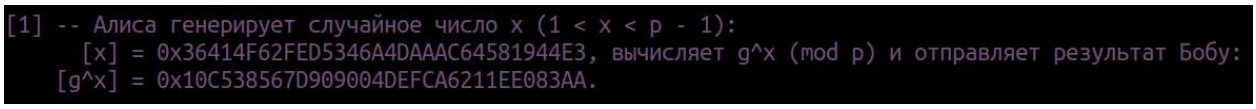

Рисунок 7 – Выполнение шага 1 протокола

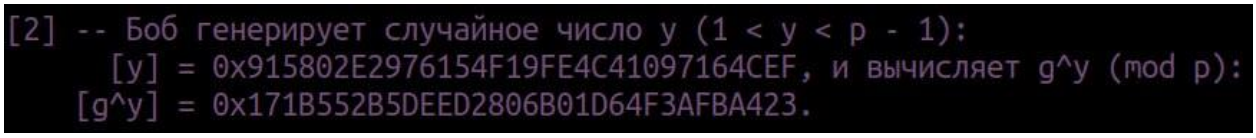

Рисунок 8 – Выполнение шага 2 протокола

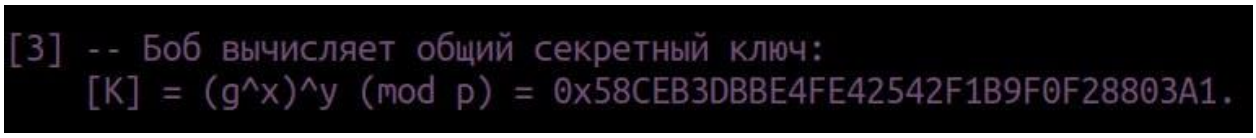

Рисунок 9 – Выполнение шага 3 протокола

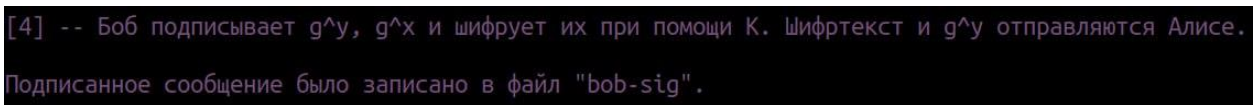

Рисунок 10 – Выполнение шага 4 протокола

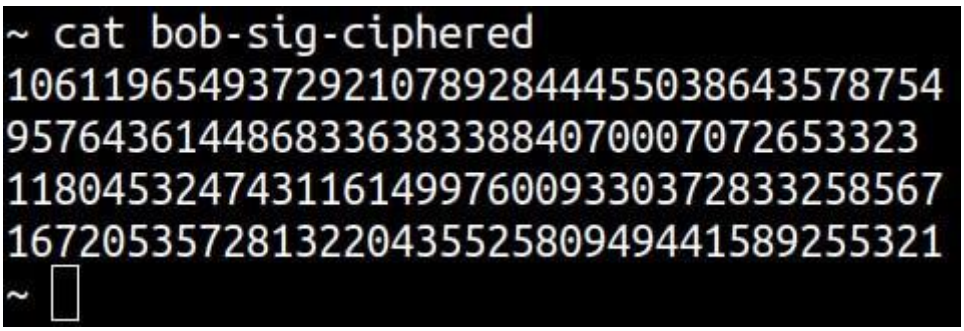

Рисунок 11 – Содержимое файла bob-sig-ciphered (зашифрованная подпись пользователя BOB)

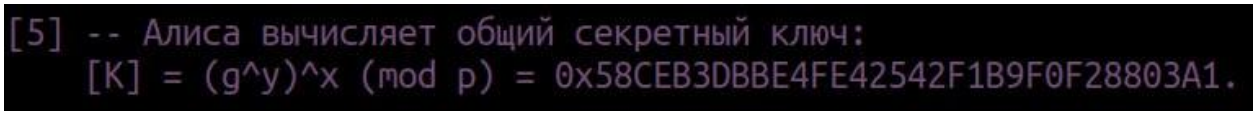

Рисунок 12 – Выполнение шага 5 протокола

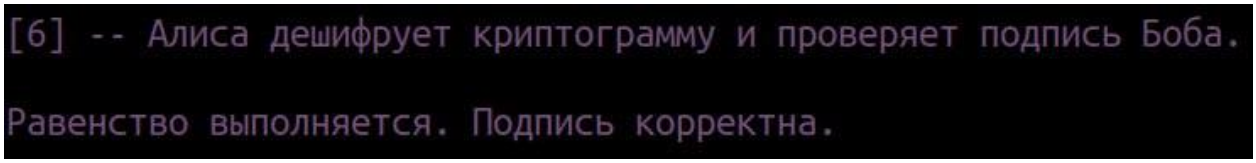

Рисунок 13 – Выполнение шага 6 протокола

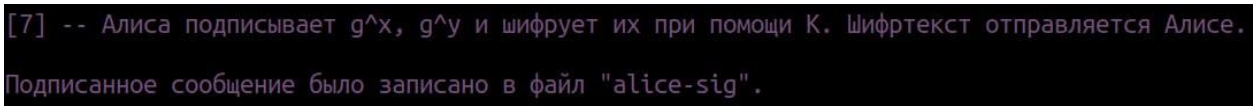

Рисунок 14 – Выполнение шага 7 протокола

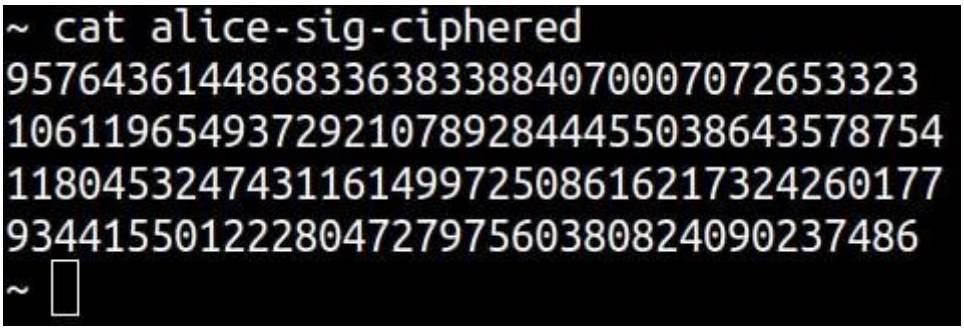

Рисунок 15 – Содержимое файла alice-sig-ciphered (зашифрованная подпись пользователя ALICE)

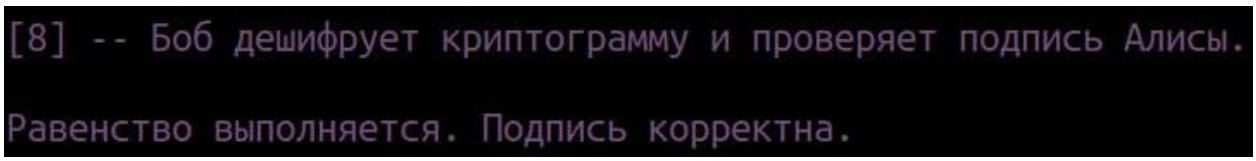

Рисунок 16 – Выполнение шага 8 протокола

Листинг программы

```
(defpackage #:ec-arith
  (:use #:cl)
  (:export #:add-points
           #:scalar-product))

(in-package #:ec-arith)



(defun add-points (P Q modulo)
  (when (eql P 'INF)
    (return-from add-points Q))
  (when (eql Q 'INF)
    (return-from add-points P))
  (let ((Px (car P)) (Py (cadr P))
        (Qx (car Q)) (Qy (cadr Q))
        (Rx nil) (Ry nil)
        (frac nil))
    (cond
      ((/= Px Qx) (setq frac (* (- Qy Py)
                                (cadr (aux:ext-gcd (mod (- Qx Px)
modulo) modulo)))
                        Rx (- (* frac frac) (+ Px Qx))
                        Ry (+ (- Py) (* frac (- Px Rx)))))
      (t (cond
           ((= Py Qy) (setq frac (* (* 3 Px Px)
```

```
                                        (cadr (aux:ext-gcd (mod (* 2 Py)
modulo)
                                                           modulo)))
                                Rx (- (* frac frac) (* 2 Px))
                                Ry (- (* frac (- Px Rx)) Py)))
            (t (return-from add-points 'INF)))))
    (list (mod Rx modulo) (mod Ry modulo))))


(defun scalar-product (n P modulo)
  (let ((result 'INF) (addend P))
    (aux:while (not (zerop n))
      (when (= 1 (mod n 2))
        (setq result (add-points addend result modulo)))
      (setq addend (add-points addend addend modulo)
            n (ash n -1)))
    result))
```

```
(defpackage #:gen-ec
  (:use #:cl)
  (:export #:generate-curve))

(in-package #:gen-ec)


(defun from-binary-to-decimal (binary-form)
  (apply #'+ (mapcar #'(lambda (bit power) (* bit (expt 2 power)))
                     binary-form
                     (loop for i from (1- (length binary-form)) downto 0
                           collect i))))


(defun find-prime-mod-6 (lower-bound upper-bound starter)
  (let ((rem (mod starter 6)) (starter-copy))
    (when (= 3 rem) (setq starter (- starter 2)))
    (when (= 5 rem) (setq starter (- starter 4)))
    (when (< starter lower-bound) (setq starter (+ 6 starter)))
    (setq starter-copy starter)
    (do ((iter starter (+ 6 iter))) ((> iter upper-bound))
      (when (aux:miller-rabin iter) (return-from find-prime-mod-6 iter)))
    (do ((iter (+ 1 lower-bound (- 6 (mod lower-bound 6))) (+ 6 iter)))
        ((> iter starter-copy))
      (when (aux:miller-rabin iter) (return-from find-prime-mod-6 iter)))))


(defun generate-prime (target-length)
  (let* ((bits-generated (- target-length 3))
         (lower-bound (from-binary-to-decimal
                       (cons 1 (append (loop for i from 0 to bits-generated
                                             collect 0) '(1)))))
         (upper-bound (from-binary-to-decimal
                       (cons 1 (append (loop for i from 0 to bits-generated
                                             collect 1) '(1)))))
         (prime? (from-binary-to-decimal
                  (cons 1 (append (loop for i from 0 to bits-generated
                                        collect (random 2)) '(1))))))
```

```
    (when (and (= 1 (mod prime? 6)) (aux:miller-rabin prime?))
      (return-from generate-prime prime?))
    (find-prime-mod-6 lower-bound upper-bound prime?)))


(defun is-power-residue (b p-char power)
  (when (or (= 2 power) (= 3 power))
    (= 1 (aux:mod-expt b (floor (1- p-char) power) p-char))))


(defun compute-legendre (a p)
  (when (zerop (mod a p))
    (return-from compute-legendre 0))
  (if (is-power-residue a p 2) 1 -1))


(defun find-k-i (a-i q p)
  (do ((k 0 (1+ k))) ((= 1 (aux:mod-expt a-i (* (expt 2 k) q) p)) k)))


(defun get-inv (a p)
  (cadr (aux:ext-gcd a p)))


(defun seq-sqrt-Zp (a p)
  (when (/= 1 (compute-legendre a p)) (return-from seq-sqrt-Zp))
  (let ((b) (k-i -1) (k-is) (r-i) (m 0) (q (1- p))
        (a-prev a) (a-cur a) (pow))
    (aux:while (zerop (logand q 1)) (setq m (1+ m) q (ash q -1)))
    (aux:while (/= -1 (compute-legendre (setq b (random p)) p)))
    (aux:while (not (zerop k-i))
      (setq  k-i    (find-k-i a-cur q p) k-is (cons k-i k-is))
      (psetq a-cur  (mod (* a-cur (aux:mod-expt b (ash 1 (- m k-i)) p)) p)
             a-prev a-cur))
    (setq k-is (cdr k-is) r-i (aux:mod-expt a-prev (ash (1+ q) -1) p))
    (do ((i (length k-is) (1- i))) ((zerop i) r-i)
      (setq pow (ash 1 (- m (car k-is) 1))
            r-i (mod (* r-i (get-inv (aux:mod-expt b pow p) p)) p)
```

```
            k-is (cdr k-is)))))


(defun compute-u (D p)
  (let* ((D-normed (mod D p))
         (root? (seq-sqrt-zp D-normed p)))
    (aux:while (/= D-normed (mod (* root? root?) p))
      (setq root? (seq-sqrt-zp D-normed p)))
    root?))


(defun get-ring-factorization (p-char &optional (D 3))
  (let ((legendre (compute-legendre (- D) p-char))
        (u-i) (iter) (u-is) (m-i) (m-is) (a-i) (b-i 1)
        (a-is) (b-is) (a-i-f-num) (a-i-s-num) (b-i-num)
        (denom))
    (when (= -1 legendre) (return-from get-ring-factorization))
    (setq u-is (cons (compute-u (- D) p-char) u-is)
          m-is (cons p-char m-is))
    (do ((i 0 (1+ i))) ((= 1 (car m-is)) (setq a-i u-i iter (1- i)))
      (setq u-i (car u-is) m-i (car m-is))
      (setq m-is (cons (/ (+ (* u-i u-i) D) m-i) m-is) m-i (car m-is)
            u-is (cons (min (mod u-i m-i) (mod (- m-i u-i) m-i)) u-is)))
    (setq u-is (cddr u-is))
    (do ((j iter (1- j))) ((zerop j) (list a-i b-i))
      (setq u-i (car u-is) a-i-f-num (* u-i a-i)
            a-i-s-num (* D b-i) b-i-num (* u-i b-i)
            denom (+ (* a-i a-i) (* D b-i b-i)))
      (psetq a-is (list (/ (+ a-i-f-num a-i-s-num) denom)
                        (/ (- a-i-f-num a-i-s-num) denom))
             b-is (list (/ (+ (- a-i) b-i-num) denom)
                        (/ (- (- a-i) b-i-num) denom)))
      (if (integerp (car a-is))
          (setq a-i (car a-is))
          (setq a-i (cadr a-is)))
      (if (integerp (car b-is))
          (setq b-i (car b-is))
          (setq b-i (cadr b-is)))
```

```
      (setq m-is (cdr m-is) u-is (cdr u-is)))))


(defun get-char-and-factors (target-length)
  (let ((p-char) (factors))
    (aux:while (null (setq p-char (generate-prime target-length)
                           factors (get-ring-factorization p-char))))
    (cons p-char factors)))


(defun get-possible-#Es (p-c-d)
  (let* ((succ-p (1+ (car p-c-d)))
         (c (cadr p-c-d)) (d (caddr p-c-d))
         (possible-s (list (+ c (* 3 d)) (- c (* 3 d)) (* 2 c))))
    (append (mapcar #'(lambda (s) (+ succ-p s)) possible-s)
            (mapcar #'(lambda (s) (- succ-p s)) possible-s))))


(defun routine (divider)
  (lambda (dividend) (zerop (mod dividend divider))))


(defun check-equalities (possible-#Es)
  (let ((divisors '(1 6 3 2)) (E-m-divisor) (m))
    (dolist (divisor divisors)
      (dolist (E# possible-#Es)
        (when (funcall (routine divisor) E#)
          (setq m (floor E# divisor))
          (when (aux:miller-rabin m)
            (setq E-m-divisor (cons (list E# m divisor) E-m-divisor))))))
    E-m-divisor))


(defun generate-P0-and-b (p-char)
  (let ((x0 0) (y0 0) (b nil))
    (aux:while (zerop x0)
      (setq x0 (random p-char)))
    (aux:while (zerop y0)
```

```
      (setq y0 (random p-char)))
    (setq b (mod (- (* y0 y0) (* x0 x0 x0)) p-char))
    (list (list x0 y0) b)))


(defun check-residues (b p-char divisor)
  (cond ((= 1 divisor) (and (not (is-power-residue b p-char 2))
                            (not (is-power-residue b p-char 3))))
        ((= 6 divisor) (and (is-power-residue b p-char 2)
                            (is-power-residue b p-char 3)))
        ((= 3 divisor) (and (is-power-residue b p-char 2)
                            (not (is-power-residue b p-char 3))))
        ((= 2 divisor) (and (not (is-power-residue b p-char 2))
                            (is-power-residue b p-char 3)))
        (t nil)))


(defun generate-curve (req-length m-sec)
  (let ((p-d-e) (p-char) (Es) (E-m-divisor)
        (P0-and-b) (generator))
    (tagbody generate-p
       (aux:while (null (setq p-d-e (get-char-and-factors req-length)
                              p-char (car p-d-e)
                              Es (get-possible-#Es p-d-e)
                              E-m-divisor (check-equalities Es))))
       (setq E-m-divisor (nth (random (length E-m-divisor)) E-m-divisor))
       (when (= (cadr E-m-divisor) p-char)
         (go generate-p))
       (dotimes (iter m-sec)
         (when (= 1 (aux:mod-expt p-char (1+ iter) (cadr E-m-divisor)))
           (go generate-p))))
    (tagbody generate-point-and-b
       (setq P0-and-b (generate-P0-and-b p-char))
       (when (not (check-residues (cadr P0-and-b) p-char (caddr E-m-divisor)))
         (go generate-point-and-b))
       (if (eql (ec-arith::scalar-product (car E-m-divisor)
                                          (car P0-and-b)
                                          p-char)
```

```
            EC-ARITH::'INF)
        (setq generator (ec-arith::scalar-product (caddr E-m-divisor)
                                                  (car P0-and-b)
                                                  p-char))
        (go generate-point-and-b)))
(list p-char (cadr P0-and-b) (cadr E-m-divisor) generator)))
```

```
(defpackage #:ec-schnorr
  (:use :cl))

(in-package #:ec-schnorr)


(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 (datum data)
      (if (atom datum)
          (format out "~a~%" datum)
          (format out "~a ~a~%" (car datum) (cadr datum))))))


(defun schnorr-generate-keys (key-length filename-prefix &optional (m-sec 30))
  (let ((l) (lQ) (pub-filename) (priv-filename))
    (destructuring-bind (p b r Q)
        (gen-ec:generate-curve key-length m-sec)
      (setq l (+ 2 (random (- r 2)))
            lQ (ec-arith::scalar-product l Q p))
      (format t "~%Значение открытого ключа проверки подписи пользователя ~s
было записано в файл ~s.~%"
              (string-upcase filename-prefix)
              (setq pub-filename (concatenate 'string filename-prefix "-sig-
pub-key")))
      (write-to-file (list p b r Q lQ) pub-filename)
      (format t "Значение секретного ключа формирования подписи пользователя
~s было записано в файл ~s.~%"
              (string-upcase filename-prefix)
              (setq priv-filename (concatenate 'string filename-prefix "-sig-
priv-key")))
      (write-to-file (list l) priv-filename))))
```

```
(defun extract (filename &optional (opt))
  (let ((extr) (len-extr) (keys))
    (handler-case (setq extr (uiop:read-file-lines filename))
      (error (err)
        (format t "В ходе программы было выполнено ошибочное условие:~%~a~%"
err)
        (return-from extract)))
    (setq extr (mapcar #'(lambda (str)
                           (uiop:split-string str :separator " ")) extr))
    (handler-case
        (setq extr (mapcar #'parse-integer (apply #'append extr)))
      (error (err)
        (format t "В ходе работы программы было выполнено ошибочное
условие:~%~a~%" err)
        (return-from extract)))
    (setq len-extr (length extr)
          keys (list (nth 0 extr) (nth 2 extr)
                     (list (nth 3 extr) (nth 4 extr))))
    (cond ((= 1 len-extr) extr)
          ((= 7 len-extr) (if (eql opt 'P-NEEDED)
                              (append keys (list (list (nth 5 extr) (nth 6
extr))))
                              keys))
          (t nil))))


(defun extract-keys (prefix &key (l-req) (p-req))
  (let ((extracted-public) (extracted-private))
    (setq extracted-public (extract (concatenate 'string prefix "-sig-pub-
key") p-req))
    (when (null l-req)
      (setq extracted-private (extract (concatenate 'string prefix "-sig-priv-
key"))))
    (append extracted-public extracted-private)))
```

```
(defun extract-message (filename)
  (let ((message))
    (handler-case (setq message (uiop:read-file-lines filename))
      (error (err)
        (format t "В ходе работы программы было выполнено ошибочное условие:~%~a~%" err)
        (return-from extract-message)))
    message))


(defun schnorr-sign-message (prefix file-message)
  (let ((k) (kQ) (message) (e) (s) (sig-filename))
    (destructuring-bind (p r Q l) (extract-keys prefix)
      (setq message (extract-message file-message))
      (tagbody try-again-k
         (while (not (< 0 (setq k (random r)) r)))
         (setq kQ (ec-arith::scalar-product k Q p)
               e (sxhash (cons kQ message)))
         (when (zerop (mod e r)) (go try-again-k)))
      (setq s (mod (+ (* l e) k) r))
      (format t "~%Подписанное сообщение было записано в файл ~s.~%"
              (setq sig-filename (concatenate 'string prefix "-sig")))
      (write-to-file (append message (list e s)) sig-filename))))


(defun extract-signature (filename)
  (let ((signature) (sig-len) (e-idx) (s-idx)
        (at-e-idx) (at-s-idx))
    (handler-case
        (setq signature (uiop:read-file-lines filename))
      (error () (return-from extract-signature)))
    (setq sig-len (length signature))
    (setq s-idx (1- sig-len) at-s-idx (nth s-idx signature)
          e-idx (1-   s-idx) at-e-idx (nth e-idx signature))
    (handler-case (setf (nth e-idx signature) (setq at-e-idx (parse-integer at-e-idx))
                        (nth s-idx signature) (setq at-s-idx (parse-integer at-s-idx)))
```

```
      (error () (return-from extract-signature)))
    (list (subseq signature 0 e-idx) at-e-idx at-s-idx)))


(defun get-minus-P (P m)
  (when (equal "INF" P) (return-from get-minus-P "INF"))
  (when (zerop (cadr P)) (return-from get-minus-P P))
  (list (car P) (- m (cadr P))))


(defun schnorr-verify-signature (prefix filename)
  (let ((R?) (e?) (keys (extract-keys prefix :l-req 'NO-L :p-req 'P-NEEDED)))
    (setq keys (cons (car keys) (subseq keys 2)))
    (destructuring-bind (p Q lQ m e s) (append keys (extract-signature
filename))
      (setq R? (ec-arith::add-points (ec-arith::scalar-product s Q p)
                                     (ec-arith::scalar-product e (get-minus-P
lQ p) p)
                                     p)
            e? (sxhash (cons R? m)))
      (if (= e? e)
          (format t "~%Равенство выполняется. Подпись корректна.~%")
          (format t "~%Равенство не выполняется. Файл с сообщением или
параметры подписи были изменены.~%")))))
```

```
(defpackage #:aux
  (:use #:cl)
  (:export #:while
           #:ext-gcd
           #:mod-expt
           #:miller-rabin))

(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 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))
  (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)))))
```

```
(defpackage #:st2st
  (:use :cl))

(in-package #:st2st)


(defun stop () (read-line))


(defun get-bit-len ()
  (let ((bit-len))
    (tagbody try-again
       (setq bit-len (read))
       (when (and (integerp bit-len) (aux::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 gen-p&g (bit-len)
  (let ((p) (g))
    (tagbody gen-prime
       (setq p (aux::generate-prime bit-len)
             g (aux::find-g p))
       (when (null g) (go gen-prime)))
    (list p g)))


(defun step-1 (p g)
  (let ((x) (g^x))
    (format t "~%[1] -- Алиса генерирует случайное число x (1 < x < p - 1):
      [x] = 0x~x, вычисляет g^x (mod p) и отправляет результат Бобу:
    [g^x] = 0x~x.~%" (setq x (+ 2 (random (- p 3)))) (setq g^x (aux::mod-expt
g x p)))
    (aux::write-to-file (list x g^x) "alice-dh-keys")
    (list x g^x)))
```

```
(defun step-2 (p g)
  (let ((y) (g^y))
    (format t "~%[2] -- Боб генерирует случайное число y (1 < y < p - 1):
      [y] = 0x~x, и вычисляет g^y (mod p):
    [g^y] = 0x~x.~%" (setq y (+ 2 (random (- p 3)))) (setq g^y (aux::mod-expt
g y p)))
    (aux::write-to-file (list y g^y) "bob-dh-keys")
    (list y g^y)))


(defun step-3 (p)
  (let* ((g^x (parse-integer (uiop:read-file-line "alice-dh-keys" :at 1)))
         (y (parse-integer (uiop:read-file-line "bob-dh-keys" :at 0)))
         (shared-key (aux::mod-expt g^x y p)))
    (format t "~%[3] -- Боб вычисляет общий секретный ключ:
    [K] = (g^x)^y (mod p) = 0x~x.~%" shared-key)
    (aux::write-to-file (list shared-key) "bob-shared-key")))


(defun encrypt (prefix fn-plaintext fn-key)
  (let*  ((plaintext  (mapcar  #'parse-integer  (uiop:read-file-lines  fn-
plaintext)))
         (key (parse-integer (uiop:read-file-line fn-key :at 0)))
         (ciphered (mapcar #'(lambda (plain) (logxor plain key)) plaintext)))
    (aux::write-to-file  ciphered  (concatenate  'string  prefix  "-sig-
ciphered"))))


(defun step-4 ()
  (let ((g^y (parse-integer (uiop:read-file-line "bob-dh-keys" :at 1)))
        (g^x (parse-integer (uiop:read-file-line "alice-dh-keys" :at 1))))
    (format t "~%[4] -- Боб подписывает g^y, g^x и шифрует их при помощи K.
Шифртекст и g^y отправляются Алисе.~%")
    (aux::write-to-file (list g^y g^x) "transient")
    (ec-schnorr::schnorr-sign-message "bob" "transient")
    (encrypt "bob" "bob-sig" "bob-shared-key")
    (uiop:run-program "rm transient bob-sig")))
```

```
(defun step-5 (p)
  (let* ((g^y (parse-integer (uiop:read-file-line "bob-dh-keys" :at 1)))
         (x (parse-integer (uiop:read-file-line "alice-dh-keys" :at 0)))
         (shared-key (aux::mod-expt g^y x p)))
    (format t "~%[5] -- Алиса вычисляет общий секретный ключ:
    [K] = (g^y)^x (mod p) = 0x~x.~%" shared-key)
    (aux::write-to-file (list shared-key) "alice-shared-key")))


(defun decrypt (fn-cipher fn-key)
  (let ((ciphertext (mapcar #'parse-integer (uiop:read-file-lines fn-cipher)))
        (key (parse-integer (uiop:read-file-line fn-key :at 0))))
    (aux::write-to-file (mapcar #'(lambda (cipher) (logxor cipher key))
                                ciphertext) "transient")))


(defun step-6 ()
  (format t "~%[6] -- Алиса дешифрует криптограмму и проверяет подпись
Боба.~%")
  (decrypt "bob-sig-ciphered" "alice-shared-key")
  (ec-schnorr::schnorr-verify-signature "bob" "transient")
  (uiop:run-program "rm transient"))


(defun step-7 ()
  (let ((g^x (parse-integer (uiop:read-file-line "alice-dh-keys" :at 1)))
        (g^y (parse-integer (uiop:read-file-line "bob-dh-keys" :at 1))))
    (format t "~%[7] -- Алиса подписывает g^x, g^y и шифрует их при помощи K.
Шифртекст отправляется Алисе.~%")
    (aux::write-to-file (list g^x g^y) "transient")
    (ec-schnorr::schnorr-sign-message "alice" "transient")
    (encrypt "alice" "alice-sig" "alice-shared-key")
    (uiop:run-program "rm transient alice-sig")))


(defun step-8 ()
```

```
  (format t "~%[8] -- Боб дешифрует криптограмму и проверяет подпись Алисы.~%")
  (decrypt "alice-sig-ciphered" "bob-shared-key")
  (ec-schnorr::schnorr-verify-signature "alice" "transient")
  (uiop:run-program "rm transient"))


(defun st2st ()
  (let ((bit-len-sig) (bit-len-sts))
    (format t "~%Введите битовую длину подписи l (l = 2^k, l > 16): ")
    (setq bit-len-sig (get-bit-len))
    (ec-schnorr::schnorr-generate-keys bit-len-sig "alice")
    (ec-schnorr::schnorr-generate-keys bit-len-sig "bob")
    (format t "~%Введите битовую длину модуля p (STS): ")
    (setq bit-len-sts (get-bit-len))
    (destructuring-bind (p g) (gen-p&g bit-len-sts)
      (format t "~%Были сгенерированы параметры:
    [p] = 0x~x;~%    [g] = 0x~x.~%" p g)
      (step-1 p g) (stop) (step-2 p g) (stop)
      (step-3 p  ) (stop) (step-4    ) (stop)
      (step-5 p  ) (stop) (step-6    ) (stop)
      (step-7    ) (stop) (step-8    ) (stop))))
```