

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

Скрытый канал связи

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

подпись, дата

Р. А. Фарахутдинов

Саратов 2023

1 Постановка задачи

Необходимо реализовать скрытый канал связи на основе ESIGN.

2 Теоретические сведения

Протокол ESIGN (Efficient digital signature – эффективная цифровая подпись) — схема цифровой подписи с открытым ключом, основанная на проблеме факторизации чисел. Цифровая подпись была разработана в японской компании NTT в 1985 году. Отличительной чертой данной схемы является возможность быстрой генерации подписи.

Основные параметры схемы. Алиса подписывает открытое сообщение m , Боб проверяет подпись. Генерируются: p, q – большие простые числа одинаковой длины; вычисляется $n = p^2q$; выбирается $k \geq 4$ (параметр безопасности). Элементы $\{n, k\}$ объявляются открытым ключом, элементы $\{p, q\}$ закрытым ключом Алисы.

Генерация подписи

1. $A: \{h\}$, где $h = H(m)$ – хэш-функция со значением от 0 до $n - 1$, этот шаг можно опустить, если сообщение удовлетворяет неравенству $0 \leq m \leq n - 1$;
2. $A: \{x\}$, где x – случайное число из интервала $0 < x < pq$;
3. $A: \{a, b\}$, где

$$a = \left\lfloor \frac{h - (x^k \bmod n)}{pq} \right\rfloor, b = a(k \cdot x^{k-1})^{-1} \bmod p$$

4. $A: \{s\}$, где $s = x + bpq$;
5. $A \rightarrow B: \{m, s\}$.

Проверка подписи

6. $B: \{h\}$, где $h = H(m)$;
7. $B: \{c\}$, где $c = \left\lceil \frac{2}{3} \log_2 n \right\rceil$;
8. B : проверяет неравенство, что $h \leq s^k \bmod n \leq h + 2^c$.

Основные параметры схемы в случае скрытого канала. Алиса подписывает безобидное сообщение m , Боб проверяет подпись. Вместе в этом передаётся секретное сообщение m^* . Генерируются: p, q, r – большие простые числа одинаковой длины; вычисляется $n = p^2qr$; выбирается $k \geq 4$ (параметр безопасности). Элементы $\{n, k\}$ объявляются открытым ключом, элементы $\{p, q, r\}$ – закрытым ключом Алисы. Часть этого ключа, а именно r , должна быть известна Бобу для извлечения секретного сообщения и необходимо выполнение неравенства $m^* < r$.

Генерация подписи

1. $A: \{h\}$, где $h = H(m)$ – хэш-функция со значением от 0 до $n - 1$;
2. $A: \{x\}$, где $x = m^* + ur$ и u – случайное число из интервала $0 < u < pq - 1$;
3. $A: \{a, b\}$, где

$$a = \left\lceil \frac{h - (x^k \bmod n)}{pqr} \right\rceil, b = a(k \cdot x^{k-1})^{-1} \bmod p;$$

4. $A: \{s\}$, где $s = x + bpqr$;
5. $A \rightarrow W(B): \{m, s\}$.

Проверка подписи

6. $W: \{h\}$, где $h = H(M)$;
7. $W: \{c\}$, где $c = \left\lceil \frac{2}{3} \log_2 n \right\rceil$;
8. W : проверяет неравенство, что $h \leq s^k \bmod n \leq h + 2^c$;
9. $W \rightarrow B: \{m, s\}$.

Получение секретного сообщения

10. B : Боб также проверяет подпись и извлекает секретное сообщение,
 $m^* = s \bmod r$.

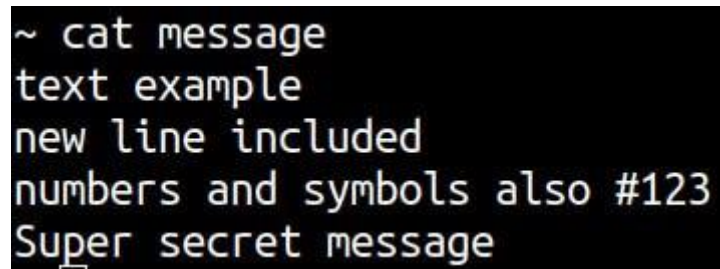
3 Практическая реализация

3.1 Описание программы

Язык программной реализации – Common Lisp. Программа реализует функции генерации ключей, подписи сообщения (без скрытого канала и с ним), проверки подписи (без скрытого канала и с ним) и извлечения скрытого сообщения. Выполнение каждой из функций разбито на шаги в соответствии с пунктами рассматриваемого протокола. Также для удобства представления все числа представлены в шестнадцатеричной системе счисления.

3.2 Результаты тестирования программы

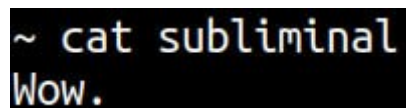
Рассмотрим работу программы для битовой длины l открытого ключа n , равной 128 и безобидного сообщения m :



```
~ cat message
text example
new line included
numbers and symbols also #123
Super secret message
```

Рисунок 1 – Безобидное сообщение m

и скрываемого сообщения m^* :



```
~ cat subliminal
Wow.
```

Рисунок 2 – Скрываемое сообщение m^*

Сгенерируем простые числа p, q, r и начнём выполнение протокола.

```

ESIGN-SC> (gen-keys)

[ЭТАП ГЕНЕРАЦИИ КЛЮЧЕЙ]

Введите битовую длину  $l$  открытого ключа  $n$  ( $l = 2^m$ ,  $l > 63$ ): 128

Были сгенерированы значения:

p = 0x88B6810B;
q = 0x8EBF361D;
r = 0xA918EFBB;
n = 0x1AE407DB5636D1BAFE704E0E55F9EA37.

```

Рисунок 3 – Результат генерации простых чисел p , q и r

```

ESIGN-SC> (sign-message-usual)

[ПОДПИСЬ ОБЫЧНОГО СООБЩЕНИЯ]

[1] -- Определение подписываемого сообщения:

Введите имя файла, в котором содержится сообщение (по умолчанию message):

[2] -- Определение параметра безопасности k:

Введите значение параметра безопасности k ( $k \geq 4$ , по умолчанию 4):

```

Рисунок 4 – Извлечение сообщения и определение параметра безопасности при обычной подписи сообщения

```

[3] -- Чтобы подписать обычное сообщение, Алиса сначала выбирает случайное число  $x$ ,
      меньшее  $pqr$ :

      x = 0x57D9DAC1222410737057DE;

[4] -- Далее она вычисляет  $w$  -- наименьшее целое, которое больше или равно
       $(H(m) - x^k \bmod n) / (pqr)$ :

      w = 0x1BF08BF;

```

Рисунок 5 – Выполнение шагов 3 и 4 этапа обычной подписи сообщения

```
[5] -- Затем она вычисляет значение  $s = x + ((w / (kx^{k-1})) \bmod p) * rqr$  подписи:  
s = 0x1350B9A41993A47A55693F73EDCB165E.
```

Рисунок 6 – Выполнение шага 6 этапа обычной подписи сообщения

После генерации подписи проверим её корректность.

```
ESIGN-SC> (verify-message-usual)  
  
[ПРОВЕРКА ПОДПИСИ ОБЫЧНОГО СООБЩЕНИЯ]  
  
[1] -- Определение проверяемого сообщения:  
Введите имя файла, в котором содержится сообщение (по умолчанию message):  
  
[2] -- Определение проверяемой подписи:  
Введите имя файла, в котором содержится подпись (по умолчанию s):
```

Рисунок 7 – Определение сообщения и подписи

```
[3] -- Для проверки подписи Боб вычисляет  $s^k \bmod n$ :  
s^k (mod n) = 0xD5E5211C0E4230A4345A879C9CADCAE;  
  
[4] -- Кроме того, он вычисляет a, наименьшее целое, которое больше или равно  
    утроенному числу битов n, делённому на четыре:  
a = 94;
```

Рисунок 8 – Выполнение шагов 3 и 4 этапа обычной проверки подписи сообщения

```
[5] -- Если  $H(m)$  меньше или равна  $s^k \bmod n$ , и если  $s^k \bmod n$  меньше  $H(m) + 2^a$ ,  
    то подпись считается правильной:  
  
H(m) = 0xD5E5211BA99B90A172AA22F65D85F50;  
s^k (mod n) = 0xD5E5211C0E4230A4345A879C9CADCAE;  
H(m) + 2^a = 0xD5E5211FA99B90A172AA22F65D85F50;  
2^a = 0x00000000400000000000000000000000.  
  
Подпись корректна.
```

Рисунок 9 – Результат обычной проверки подписи сообщения

Теперь проверим работу функций с использованием скрытого канала.

```
ESIGN-SC> (sign-message-subliminal)

[ПОДПИСЬ СООБЩЕНИЯ С ВНЕДРЕНИЕМ СКРЫТОГО]

[1] -- Ввод "безобидного" сообщения:
Введите имя файла, в котором содержится сообщение (по умолчанию message):

[2] -- Ввод скрываемого сообщения:
Введите имя файла, в котором содержится сообщение (по умолчанию message): subliminal
```

Рисунок 10 – Определение безобидного и скрываемого сообщений

```
[3] -- Чтобы скрыть сообщение M, используя сообщение M', Алиса вычисляет
       $x' = M + ug$  (u из  $[1, pq - 1]$ ):

      E(M) = 0x576F772E;
      x'    = 0x31312A7C20E8A5265539A8F4;

[4] -- Далее она вычисляет w как наибольшее целое выражения
       $(H(M') - x'^k \pmod n) / (pqg)$ :

      w' = 0x4063DE60;
```

Рисунок 11 – Выполнение шагов 3 и 4 этапа подписи сообщения со скрытым каналом

```
[5] -- Затем Алиса аналогично вычисляет s с использованием модифицированных x' и w':

      s = 0xD609212F7974AF074CFD111F9C0CB3A.
```

Рисунок 12 – Результат генерации подписи со скрытым каналом

Проверим корректность подписи с использованием скрытого канала.

```
ESIGN-SC> (verify-message-subliminal)

[ПРОВЕРКА ПОДПИСИ СО СКРЫТЫМ СООБЩЕНИЕМ]

[1] -- Определение проверяемого сообщения:
Введите имя файла, в котором содержится сообщение (по умолчанию message):

[2] -- Определение проверяемой подписи:
Введите имя файла, в котором содержится подпись (по умолчанию s): s-sublim
```

Рисунок 13 – Определение сообщение и подписи

```
[3] -- Уолтер вычисляет  $s^k \pmod n$ :

 $s^k \pmod n = 0xD5E5211CD4984152A735303404F2434.$ 

[4] -- Он убеждается, что подпись корректна путём проверки равенства
 $H(m) \leq s^k \pmod n < H(m) + 2^a$ :

H(M)          = 0xD5E5211BA99B90A172AA22F65D85F50;
 $s^k \pmod n$     = 0xD5E5211CD4984152A735303404F2434;
H(M) +  $2^a$      = 0xD5E5211FA99B90A172AA22F65D85F50;
 $2^a$           = 0x00000004000000000000000000000000.

Подпись корректна.
```

Рисунок 14 – Результат проверки подписи со скрытым каналом

Теперь извлечём скрытое сообщение из подписи.

```
ESIGN-SC> (recover-subliminal-message)

[ИЗВЛЕЧЕНИЕ СКРЫТОГО СООБЩЕНИЯ ИЗ ПОДПИСИ]

[1] -- Определение файла с подписью:
Введите имя файла, в котором содержится подпись (по умолчанию s): s-sublim
```

Рисунок 15 – Определение подписи


```
[2] -- Для восстановления скрытого сообщения достаточно вычислить
       $s = x' + urqg = M + ug + urqg = M \pmod{r}$ :

       $E(M) = 0x576F772E$ ;

[3] -- Расшифруем  $E(M)$  и получим:

       $M = \text{"Wow."}$ .
```

Рисунок 16 – Результат извлечения сообщения из подписи со скрытым каналом

Листинг программы

```
(defpackage #:aux
  (:use :cl)
  (:export #:write-to-file
           #:read-parse
           #:mod-expt
           #:miller-rabin
           #:ext-gcd
           #:generate-prime))

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

(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 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))))
```

```
(defpackage :crypt
  (:use #:cl)
  (:export #:aes-encrypt
           #:aes-decrypt
           #:hash))
```

```
(in-package :crypt)
```

```
(defun ripemd128 (str)
  (ironclad:byte-array-to-hex-string
   (ironclad:digest-sequence
    :ripemd-128
    (ironclad:ascii-string-to-byte-array str))))
```

```
(defun get-cipher (key)
  (ironclad:make-cipher :aes
    :mode :ecb
    :key (ironclad:ascii-string-to-byte-array (ripemd128 key))))
```

```
(defun aes-encrypt (plaintext key)
  (let ((cipher (get-cipher key))
        (msg (ironclad:ascii-string-to-byte-array plaintext)))
    (ironclad:encrypt-in-place cipher msg)
    (ironclad:octets-to-integer msg)))
```

```
(defun aes-decrypt (ciphertext-int key)
  (let ((cipher (get-cipher key))
        (msg (ironclad:integer-to-octets ciphertext-int)))
    (ironclad:decrypt-in-place cipher msg)
    (coerce (mapcar #'code-char (coerce msg 'list)) 'string)))
```

```
(defun skein1024 (str)
  (ironclad:byte-array-to-hex-string
   (ironclad:digest-sequence
    :skein1024
    (ironclad:ascii-string-to-byte-array str))))
```

```
(defun hash (str modulo)
  (let ((digest (skein1024 str)))
    (mod (ironclad:octets-to-integer
          (ironclad:ascii-string-to-byte-array digest)) modulo)))
```

```
(defpackage #:sc-aux
  (:use #:cl)
  (:export #:get-bit-len
           #:gen-n
           #:concat
           #:get-message
           #:get-signature
           #:get-k
           #:pick-x
           #:compute-w
           #:compute-s
           #:compute-s^k
           #:compute-a
           #:pick-x-subliminal
           #:compute-w-subliminal
           #:compute-s-subliminal
           #:compute-s^k-subliminal))
```

```
(in-package #:sc-aux)
```

```
(defun get-bit-len ()
  "Функция считывания битовой длины числа."
  (let ((bit-len))
    (tagbody try-again
      (setq bit-len (read))
      (unless (and (integerp bit-len) (> bit-len 63)
                  (zerop (logand bit-len (1- bit-len)))))
      (format t "~%Некорректное значение битовой длины l! Попробуйте ввести
l снова: ")
      (go try-again))) bit-len))
```

```
(defun gen-n (target-len)
  "Функция генерации открытого ключа n."
  (let* ((1/4-len (ash target-len -2)) (p? (aux:generate-prime 1/4-len))
        (q? (aux:generate-prime 1/4-len)) (r? (aux:generate-prime 1/4-len))
        (n))
    (destructuring-bind (p q r) (sort (list p? q? r?) #'<))
      (aux:write-to-file (list p) "p")
      (aux:write-to-file (list q) "q")
      (aux:write-to-file (list r) "r")
      (setq n (* p p q r)))
    (aux:write-to-file (list n) "n") n))
```

```
(defun concat (message)
  (reduce #'(lambda (f s) (concatenate 'string f s)) message))
```

```
(defun get-message ()
  "Функция считывания сообщения из файла."
  (format t "~%Введите имя файла, в котором содержится сообщение (по умолчанию
message): ")
  (let ((filename))
    (tagbody try-again
      (setq filename (read-line))
      (when (zerop (length filename))
        (setq filename "message"))
      (when (not (uiop:file-exists-p filename))
        (format t "~%Файла с указанным именем не существует! Попробуйте ввести
имя файла снова: ")
        (go try-again)))
      (uiop:read-file-lines filename)))
```

```
(defun get-signature ()
  (format t "~%Введите имя файла, в котором содержится подпись (по умолчанию
s): ")
  (let ((filename))
    (tagbody try-again
      (setq filename (read-line))
      (when (zerop (length filename))
        (setq filename "s"))
      (when (not (uiop:file-exists-p filename))
        (format t "~%Файла с указанным именем не существует! Попробуйте ввести
имя файла снова: ")
        (go try-again)))
      (aux:read-parse filename)))
```

```
(defun get-k ()
  "Функция считывания параметра безопасности k."
  (let ((k) (default 4))
    (format t "~%Введите значение параметра безопасности k (k >= 4, по умолчанию
~d): "
            default)
    (tagbody try-again
      (setq k (read-line))
      (when (zerop (length k))
        (aux:write-to-file (list default) "k")
        (return-from get-k default))
      (setq k (parse-integer k :junk-allowed t))
      (when (or (null k) (< k 4))
        (format t "~%Некорректное значение k! Попробуйте ввести k снова: ")
        (go try-again)))
    (aux:write-to-file (list k) "k") k))
```

```

(defun pick-x ()
  "Подпись обычного сообщения.
  Функция выбора случайного числа x, меньшего pqr."
  (let* ((p (aux:read-parse "p")) (q (aux:read-parse "q"))
        (r (aux:read-parse "r")) (x))
    (setq x (+ 2 (random (- (* p q r) 2))))
    (aux:write-to-file (list x) "x") x))

(defun compute-w (message)
  "Подпись обычного сообщения.
  Функция вычисления w -- наименьшего целого, которое больше или равно
   $(H(m) - x^k \bmod n) / (pqr)$ ."
  (setq message (reduce #'(lambda (f s) (concatenate 'string f s)) message))
  (let ((x (aux:read-parse "x")) (k (aux:read-parse "k"))
        (n (aux:read-parse "n")) (p (aux:read-parse "p"))
        (q (aux:read-parse "q")) (r (aux:read-parse "r")) (w))
    (setq w (ceiling (/ (- (crypt:hash message n) (aux:mod-expt x k n))
                        (* p q r))))
    (aux:write-to-file (list w) "w") w))

(defun compute-s ()
  "Подпись обычного сообщения.
  Функция вычисления  $s = x + (w / (kx^{\{k-1\}} \bmod p) * pqr$  -- подписи
  сообщения m."
  (let ((x (aux:read-parse "x")) (w (aux:read-parse "w")) (k (aux:read-parse
"k"))
        (p (aux:read-parse "p")) (q (aux:read-parse "q")) (r (aux:read-parse
"r"))
        (inv) (s))
    (setq inv (mod (cadr (aux:ext-gcd (* k (aux:mod-expt x (1- k) p)) p)) p))
    (s (+ x (* (mod (* w inv) p) p q r)))
    (aux:write-to-file (list s) "s") s))

(defun compute-s^k (s)
  "Проверка подписи.
  Функция вычисления  $s^k \bmod n$ ."
  (let ((k (aux:read-parse "k")) (n (aux:read-parse "n")) (s^k))
    (setq s^k (aux:mod-expt s k n))
    (aux:write-to-file (list s^k) "s^k") s^k))

(defun compute-a ()
  "Проверка подписи.
  Функция вычисления a -- наименьшего целого, которое больше или равно

```



```

    утроенному числу битов n, делённому на четыре."
    (let ((n-bits (length (write-to-string (aux:read-parse "n") :base 2))) (a))
      (setq a (ceiling (* 3/4 n-bits)))
      (aux:write-to-file (list a) "a") a))

(defun pick-x-subliminal (subliminal-message)
  (let* ((r (aux:read-parse "r")) (n (aux:read-parse "n")) (p) (q)
         (session-key (crypt::ripemd128 (write-to-string (random n))))
         (encrypted (crypt:aes-encrypt subliminal-message session-key))
         (x-sublim))
    (when (> encrypted r)
      (format t "~%Скрытое сообщение должно быть меньше r! Завершение протокола.")
      (return-from pick-x-subliminal nil))
    (setq p (aux:read-parse "p") q (aux:read-parse "q"))
    (setq x-sublim (+ encrypted (* (1+ (random (1- (* p q)))) r)))
    (aux:write-to-file (list session-key) "subliminal-key")
    (aux:write-to-file (list encrypted) "subliminal-message-encrypted")
    (aux:write-to-file (list x-sublim) "x-sublim") x-sublim))

(defun compute-w-subliminal (innocuous-message)
  (let* ((x (aux:read-parse "x-sublim")) (k (aux:read-parse "k"))
         (n (aux:read-parse "n")) (p (aux:read-parse "p"))
         (q (aux:read-parse "q")) (r (aux:read-parse "r"))
         (w-sublim))
    (setq w-sublim (ceiling (/ (- (crypt:hash innocuous-message n)
                                   (aux:mod-expt x k n))
                               (* p q r))))
    (aux:write-to-file (list w-sublim) "w-sublim") w-sublim))

(defun compute-s-subliminal ()
  (let ((x (aux:read-parse "x-sublim")) (w (aux:read-parse "w-sublim"))
        (k (aux:read-parse "k")) (p (aux:read-parse "p"))
        (q (aux:read-parse "q")) (r (aux:read-parse "r")) (inv) (s))
    (setq inv (mod (cadr (aux:ext-gcd (* k (aux:mod-expt x (1- k) p)) p)) p))
    (setq s (+ x (* (mod (* w inv) p) p q r)))
    (aux:write-to-file (list s) "s-sublim") s))

(defun compute-s^k-subliminal (s)
  (let ((k (aux:read-parse "k")) (n (aux:read-parse "n")) (s^k-sublim))
    (setq s^k-sublim (aux:mod-expt s k n))
    (aux:write-to-file (list s^k-sublim) "s^k-sublim") s^k-sublim))

```

```
(defpackage #:esign-sc
  (:use #:cl)
  (:export #:gen-keys
           #:sign-message-usual
           #:verify-message-usual
           #:sign-message-subliminal
           #:verify-message-subliminal
           #:recover-subliminal-message))
```

```
(in-package #:esign-sc)
```

```
(defun stop () (read-line))
```

```
(defun gen-keys ()
  (format t "~%~20t[ЭТАП ГЕНЕРАЦИИ КЛЮЧЕЙ]")
  "Функция генерации открытого ( $n = pqr$ ) и закрытого ключей ( $p$ ,  $q$  и  $r$ ).
  После генерации каждое из чисел по отдельности лежит в соответствующих
  именных файлах."
  (format t "~2%Введите битовую длину  $l$  открытого ключа  $n$  ( $l = 2^m$ ,  $l > 63$ ):")
  ")
  (let* ((bit-len (sc-aux:get-bit-len)) (n) (p) (q) (r))
    (setq n (sc-aux:gen-n bit-len) p (aux:read-parse "p")
          q (aux:read-parse "q") r (aux:read-parse "r"))
    (format t "~%Были сгенерированы значения: ")
    (format t "~2%~4tp = 0x~x;~%~4tq = 0x~x;~%~4tr = 0x~x;~%~4tn = 0x~x." p q
      r n) t))
```

```
(defun sign-message-usual ()
  "Функция подписания обычного сообщения. Результатом работы функции является
  значение подписи s."
  (format t "~%~20t[ПОДПИСЬ ОБЫЧНОГО СООБЩЕНИЯ]~%")
  (format t "~%[1] -- Определение подписываемого сообщения:~%")
  (let ((message (sc-aux:get-message)) (x) (w) (s))
    (format t "~2%[2] -- Определение параметра безопасности k:~%")
    (sc-aux:get-k)
    (format t "~2%[3] -- Чтобы подписать обычное сообщение, Алиса сначала
  выбирает случайное число x,
  меньшее pqr: ")
    (setq x (sc-aux:pick-x))
    (format t "~2%~4tx = 0x~x;~%" x) (stop)
    (format t "~%[4] -- Далее она вычисляет w -- наименьшее целое, которое
  больше или равно
   $(H(m) - x^k \bmod n) / (pqr)$ : ")
    (setq w (sc-aux:compute-w message))
    (format t "~2%~4tw = 0x~x;~%" w) (stop))
```

```

(format t "~%[5] -- Затем она вычисляет значение  $s = x + ((w / (kx^{k-1})) \bmod p) * pqr$  подписи: ")
(setq s (sc-aux:compute-s))
(format t "~2%~4ts = 0x~x.~%" s) t))

```

```

(defun step-3-verify-message-usual (s)
  (let ((s^k))
    (format t "~2%[3] -- Для проверки подписи Боб вычисляет  $s^k \bmod n$ : ")
    (setq s^k (sc-aux:compute-s^k s))
    (format t "~2%~4ts^k (mod n) = 0x~x;~%" s^k) t))

```

```

(defun step-4-5-verify-message-usual (message)
  (let* ((n (aux:read-parse "n")) (digest (crypt:hash message n))
        (s^k (aux:read-parse "s^k")) (a) (2^a) (digest+) (max-len))
    (format t "~%[4] -- Кроме того, он вычисляет a, наименьшее целое, которое
    больше или равно
    утроенному числу битов n, делённому на четыре: ")
    (setq a (sc-aux:compute-a) 2^a (ash 1 a) digest+ (+ digest 2^a)
          max-len (apply #'max (mapcar #'(lambda (num)
                                            (length (write-to-string num :base
16))))
                    (list digest s^k digest+))))
    (format t "~2%~4ta = ~d;~%" a) (stop)
    (format t "~%[5] -- Если  $H(m)$  меньше или равна  $s^k \bmod n$ , и если  $s^k \bmod n$ 
    меньше  $H(m) + 2^a$ ,
    то подпись считается правильной: ")
    (format t "~2%~4tH(m) = 0x~v, '0x;~%~4ts^k (mod n) = 0x~v, '0x;~%~4tH(m)
+ 2^a = 0x~v, '0x;~%~4t2^a = 0x~v, '0x."
          max-len digest max-len s^k max-len digest+ max-len 2^a)
    (format t (if (< digest (1+ s^k) (1+ digest+))
                  "~2%Подпись корректна.~%"
                  "~2%Значение подписи некорректно.~%")) t))

```

```

(defun verify-message-usual ()
  (format t "~%~20t[ПРОВЕРКА ПОДПИСИ ОБЫЧНОГО СООБЩЕНИЯ]~%")
  (format t "~%[1] -- Определение проверяемого сообщения:~%")
  (let* ((message (sc-aux:get-message)) (s))
    (format t "~2%[2] -- Определение проверяемой подписи:~%")
    (setq s (sc-aux:get-signature))
    (step-3-verify-message-usual s) (stop)
    (step-4-5-verify-message-usual (sc-aux:concat message)) t))

```

```

(defun step-3-sign-message-subliminal (subliminal-message)
  (let ((x-sub) (encrypted))

```

```

(format t "~2%[3] -- Чтобы скрыть сообщение M, используя сообщение M',
Алиса вычисляет
  x' = M + ur (u из [1, pq - 1]): ")
(setq x-sub (sc-aux:pick-x-subliminal subliminal-message)
  encrypted (aux:read-parse "subliminal-message-encrypted"))
(when (null x-sub)
  (return-from step-3-sign-message-subliminal nil))
(format t "~2%~4tE(M) = 0x~x;~%~4tx'   = 0x~x;~%" encrypted x-sub) t))

(defun step-4-sign-message-subliminal (innocuous-message)
  (let ((w-sub))
    (format t "~%[4] -- Далее она вычисляет w как наибольшее целое выражения
      (H(M') - x'^k (mod n)) / (pqr): ")
    (setq w-sub (sc-aux:compute-w-subliminal innocuous-message))
    (format t "~2%~4tw' = 0x~x;~%" w-sub) t))

(defun step-5-sign-message-subliminal ()
  (let ((s-sub))
    (format t "~%[5] -- Затем Алиса аналогично вычисляет s с использованием
модифицированных x' и w': ")
    (setq s-sub (sc-aux:compute-s-subliminal))
    (format t "~2%~4ts   = 0x~x.~%" s-sub)))

(defun sign-message-subliminal ()
  (format t "~%~20t[ПОДПИСЬ СООБЩЕНИЯ С ВНЕДРЕНИЕМ СКРЫТОГО]~%")
  (let ((innocuous-message) (subliminal-message))
    (format t "~%[1] -- Ввод \"безобидного\" сообщения: ~%")
    (setq innocuous-message (sc-aux:concat (sc-aux:get-message)))
    (format t "~2%[2] -- Ввод скрываемого сообщения: ~%")
    (setq subliminal-message (sc-aux:concat (sc-aux:get-message)))
    (when (null (step-3-sign-message-subliminal subliminal-message))
      (return-from sign-message-subliminal nil)) (stop)
    (step-4-sign-message-subliminal innocuous-message) (stop)
    (step-5-sign-message-subliminal) t))

(defun step-3-verify-message-subliminal (s)
  (let ((s^k))
    (format t "~2%[3] -- Уолтер вычисляет s^k (mod n):")
    (setq s^k (sc-aux:compute-s^k-subliminal s))
    (format t "~2%~4ts^k (mod n) = 0x~x.~%" s^k) t))

(defun step-4-verify-message-subliminal (message)
  (format t "~%[4] -- Он убеждается, что подпись корректна путём проверки
равенства

```

```

      H(m) <= s^k (mod n) < H(m) + 2^a:~%)
(let* ((n (aux:read-parse "n")) (digest (crypt:hash message n))
      (s^k (aux:read-parse "s^k-sublim")) (a (sc-aux:compute-a))
      (2^a (ash 1 a)) (digest+ (+ digest 2^a))
      (max-len (apply #'max (mapcar #'(lambda (num)
                                          (length (write-to-string num :base
16))))
                                          (list digest s^k digest+))))
  (format t "~%~4tH(M) = 0x~v,'0x;~%~4ts^k (mod n) = 0x~v,'0x;
~4tH(M) + 2^a = 0x~v,'0x;~%~4t2^a = 0x~v,'0x.~%"
          max-len digest max-len s^k max-len digest+ max-len 2^a)
  (format t (if (< digest (1+ s^k) (1+ digest+))
                "~%Подпись корректна.~%"
                "~%Значение подписи некорректно.~%")) t))

(defun verify-message-subliminal ()
  (format t "~%~20t[ПРОВЕРКА ПОДПИСИ СО СКРЫТЫМ СООБЩЕНИЕМ]~%")
  (format t "~%[1] -- Определение проверяемого сообщения:~%")
  (let ((message (sc-aux:concat (sc-aux:get-message))) (s))
    (format t "~%[2] -- Определение проверяемой подписи:~%")
    (setq s (sc-aux:get-signature))
    (step-3-verify-message-subliminal s) (stop)
    (step-4-verify-message-subliminal message) t))

(defun recover-subliminal-message ()
  (format t "~%~20t[ИЗВЛЕЧЕНИЕ СКРЫТОГО СООБЩЕНИЯ ИЗ ПОДПИСИ]~%")
  (format t "~%[1] -- Определение файла с подписью:~%")
  (let* ((r (aux:read-parse "r")) (s (sc-aux:get-signature))
        (session-key (uiop:read-file-line "subliminal-key"))
        (encrypted (mod s r)) (decrypted))
    (format t "~%[2] -- Для восстановления скрытого сообщения достаточно
вычислить
      s = x' + ypqr = M + ur + ypqr = M (mod r): ")
    (format t "~%~4tE(M) = 0x~x;~%" encrypted) (stop)
    (setq decrypted (crypt:aes-decrypt encrypted session-key))
    (format t "~%[3] -- Расшифруем E(M) и получим:~%")
    (format t "~%~4tM = ~s.~%" decrypted) t))

```