Пшенко Артем ФИТ 3-4

Отчет по лабораторной работе № 12

Исследование алгоритмов генерации и верификации электронной цифровой подписи (Эль-Гамаль, Шнорр, RSA)

Информационная безопасность

Вариант 9

Цель: изучение алгоритмов генерации и верификации электронной цифровой подписи и приобретение практических навыков их реализации.

**А) ЭЦП Эль-Гамаля.** Ключевая информация отправителя: открытый ключ: *y*, *g* и *р*; тайный ключ: *х*. Чтобы подписать сообщение *Мо*, обладатель используемых для ЭЦП ключей должен выбрать, как и в предыдущей схеме, случайное число *k*, взаимно простое с (*р* – 1). Затем вычисляется числа *а* и *b*, являющиеся цифровой подписью (*S* = {*a*, *b*}):

*a ≡ gk*mod *p*.

Для вычисления *b* с помощью расширенного алгоритма Евклида решается уравнение:

*Н*(*Mо*) ≡ (*x \* a* + *k \* b*) mod (*p* – 1).

Получателю отправляется сообщение *М*' = *Мо* || *S*. Для верификации подписи вычисляется хеш полученного сообщения *Н*(*Мп*) = *h*. Далее нужно убедиться, что выполняется равенство:

*yaab* ≡ *gh* mod *p*.

Если равенство выполняется, подпись верифицируется.

Исходный код создания ЭЦП с помощью алгоритма Эль-Гамаля приведен в листинге 1.

Листинг 1. Исходный код создания ЭЦП с помощью алгоритма Эль-Гамаля

|  |
| --- |
| import random  import hashlib  from sympy import mod\_inverse  from math import gcd  def generate\_keys(p, g, x):  y = pow(g, x, p)  return y  def sign\_message(p, g, x, message):  while True:  k = random.randint(1, p-2) # случайное число k, взаимно простое с (р – 1)  a = pow(g, k, p) ## a = g^k mod p  h = int(hashlib.sha256(message.encode('utf-8')).hexdigest(), 16) #вычисление хеша полученного сообщ  if gcd(k, p-1) == 1:  inverse\_k = mod\_inverse(k, p-1)  if inverse\_k is not None:  break  b = ((h - x\*a) \* inverse\_k) % (p-1) ## Н(Mо) = (x^a + k^b) mod (p – 1) - отсюда ищем b  return a, b  def verify\_signature(p, g, y, a, b, message):  h = int(hashlib.sha256(message.encode('utf-8')).hexdigest(), 16) #вычисление хеша полученного сообщ  if (pow(y, a, p) \* pow(a, b, p)) % p == pow(g, h, p): ## y^a \* a^b ≡ g^h mod p.  return True  else:  return False  p = 23  g = 5  # тайный ключ  x = 6  message = 'Hello, world!'  y = generate\_keys(p, g, x)  a, b = sign\_message(p, g, x, message)  print('Подпись верна:', verify\_signature(p, g, y, a, b, message)) |

Результат работы данного алгоритма приведен на рисунке 1.



Рисунок 2. Результат работы программы

**Б) ЭЦП RSA.** Цифровая подпись S вычисляется на основе известного соотношения:

, *dо* и *no* – элементы тайного ключа отправителя. Передаваемое сообщение *М' = Мo || S*.

Соответственно, операция расшифрования на приемной стороне (получатель анализирует *Мп* || *S*) будет производиться в соответствии с формулой с известной модификацией ключей:



Далее вычисляется *Н*(*Mп*). Если *H(Mo)* = *H(Mп)*, подпись верифицирована.

Исходный код создания ЭЦП с помощью алгоритма RSA приведен в листинге 2.

Листинг 2. Исходный код создания ЭЦП с помощью алгоритма RSA

|  |
| --- |
| from cryptography.hazmat.primitives import hashes  from cryptography.hazmat.primitives.asymmetric import padding  from cryptography.hazmat.primitives.asymmetric import rsa  def generate\_keys():  private\_key = rsa.generate\_private\_key(  public\_exponent=65537,  key\_size=2048  )  public\_key = private\_key.public\_key()  return private\_key, public\_key  def sign(private\_key, message):  signature = private\_key.sign(  message,  padding.PSS(  mgf=padding.MGF1(hashes.SHA256()),  salt\_length=padding.PSS.MAX\_LENGTH  ),  hashes.SHA256()  )  return signature  def verify(public\_key, message, signature):  try:  public\_key.verify(  signature,  message,  padding.PSS(  mgf=padding.MGF1(hashes.SHA256()),  salt\_length=padding.PSS.MAX\_LENGTH  ),  hashes.SHA256()  )  print("Подпись верна.")  except:  print("Подпись неверна.")  private\_key, public\_key = generate\_keys()  message = b"Hello, World!"  signature = sign(private\_key, message)  verify(public\_key, message, signature) |

Результат работы данного алгоритма приведен на рисунке 2.



Рисунок 2. Результат работы программы

**D) ЭЦП Шнорра.** Ключевая информация: *p* – простое число в диапазоне от 512 до 1024 битов; *q* –160-битное простое число, делитель (p – 1); любое число g (g ≠ 1) такое, что *gq* ≡ 1 mod *p*.

Выбирается число *х* < *q* (*х* является тайным ключом) и вычисляется последний элемент открытого ключа: *y* ≡ *g–х* mod *p*.

Для подписи сообщения *Мо*выбирается случайное число *k* (1 < k < q) и вычисляет параметр а: *а* ≡ *gk* mod *p*.

Далее вычисляется хеш от конкатенации сообщения *Мо* и числа *а*: *h* = *H*(*Mo*||*a*). Создается хеш-образ подписываемого сообщения, спереди присоединенного к числу *а*. Далее вычисляется значение *b*: *b* ≡ (*k* + *x ⋅ h*) mod *q*.

Получателю отправляются *М'* = *Мо || S*; *S* = {*h*, *b*}.

Для проверки подписи получатель вычисляет *Х* ≡ *gbyh*(mod *p*).

Затем он проверяет выполнение равенства: *h* = *Н*(*Mп*||*Х*). Подпись достоверна, если равенство выполняется.

Исходный код создания ЭЦП с помощью алгоритма Шнорра приведен в листинге 3.

Листинг 3. Исходный код создания ЭЦП с помощью алгоритма Шнорра

|  |
| --- |
| import random  import hashlib  def generate\_keys(p, q, g):  x = random.randint(1, q - 1) # Секретный ключ  y = pow(g, -x, p) # Открытый ключ  return x, y  def sign\_message(message, p, g, q, x):  k = random.randint(1, q - 1)  a = pow(g, k, p)  h = int(hashlib.sha256(message.encode()).hexdigest(), 16)  b = (k + x \* h) % q  return a, b  def verify\_signature(message, signature, p, g, q, y):  a, b = signature  h = int(hashlib.sha256(message.encode()).hexdigest(), 16)  X = (pow(g, b, p) \* pow(y, h, p)) % p  return h == int(hashlib.sha256(message.encode()).hexdigest(), 16)  p = 30742678177050758012328021006684056407581748096353607315785686396268369844541046244025858051614268242640829910101566187864010837215586149308591486212906265025210861645801495924570466199893458505652964129877057402931556824569860418510630361272277258006813328762616228637935038757673489475989263124170036446902135012380091816705643109858212224625017571773861209415918815842504081881208437428042347805491205784569725463830578602703523615634885763893334125012037835628400181391440275233097944469221605906257918966567042606685993109980802300702642510344714857536947400772589505726815579341318529351253125292119875916918191  q = 423173346644379168618504485520574982089019682524  g = 1069841001377870851325076399414122282681104102375498133122359642376811586257803742019941805637168222909713627151758582815442501897682731861379032060410647231706446340011764580846943022642712104230896329359423795276196866768456921478582263039400215704374549719810881219496723572514667558428621707441953210467276721746575641254875577277231272204156351647568654912351297151974581139012669066258222315312380023458846763780592565433948267426465541656921274568836700140592877783505476503604876413680415203741392520817340744169831886604958943374012369325814630867321120293734748691453032793324984241378688409674730663633  x, y = generate\_keys(p, q, g)  message = "Hello, world!"  signature = sign\_message(message, p, g, q, x)  verification\_result = verify\_signature(message, signature, p, g, q, y)  print("Подпись верна:", verification\_result) |

Результат работы данного алгоритма приведен на рисунке 3.



Рисунок 3. Результат работы программы

Таким образом, в данной лабораторной работе были изучены и реализованы алгоритмы RSA. Эль-Гамаля и Шнорра для создания электронной цифровой подписи.