# Authors:

# Trevor Perrin

# Martin von Loewis - python 3 port

# Yngve Pettersen (ported by Paul Sokolovsky) - TLS 1.2

#

# See the LICENSE file for legal information regarding use of this file.

"""cryptomath module

This module has basic math/crypto code."""

from \_\_future\_\_ import print\_function

import os

import math

import base64

import binascii

from .compat import \*

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Load Optional Modules

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Try to load M2Crypto/OpenSSL

try:

from M2Crypto import m2

m2cryptoLoaded = True

except ImportError:

m2cryptoLoaded = False

#Try to load GMPY

try:

import gmpy

gmpyLoaded = True

except ImportError:

gmpyLoaded = False

#Try to load pycrypto

try:

import Crypto.Cipher.AES

pycryptoLoaded = True

except ImportError:

pycryptoLoaded = False

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# PRNG Functions

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Check that os.urandom works

import zlib

length = len(zlib.compress(os.urandom(1000)))

assert(length > 900)

def getRandomBytes(howMany):

b = bytearray(os.urandom(howMany))

assert(len(b) == howMany)

return b

prngName = "os.urandom"

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Simple hash functions

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import hmac

import hashlib

def MD5(b):

return bytearray(hashlib.md5(compat26Str(b)).digest())

def SHA1(b):

return bytearray(hashlib.sha1(compat26Str(b)).digest())

def HMAC\_MD5(k, b):

k = compatHMAC(k)

b = compatHMAC(b)

return bytearray(hmac.new(k, b, hashlib.md5).digest())

def HMAC\_SHA1(k, b):

k = compatHMAC(k)

b = compatHMAC(b)

return bytearray(hmac.new(k, b, hashlib.sha1).digest())

def HMAC\_SHA256(k, b):

k = compatHMAC(k)

b = compatHMAC(b)

return bytearray(hmac.new(k, b, hashlib.sha256).digest())

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Converter Functions

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

def bytesToNumber(b):

total = 0

multiplier = 1

for count in range(len(b)-1, -1, -1):

byte = b[count]

total += multiplier \* byte

multiplier \*= 256

return total

def numberToByteArray(n, howManyBytes=None):

"""Convert an integer into a bytearray, zero-pad to howManyBytes.

The returned bytearray may be smaller than howManyBytes, but will

not be larger. The returned bytearray will contain a big-endian

encoding of the input integer (n).

"""

if howManyBytes == None:

howManyBytes = numBytes(n)

b = bytearray(howManyBytes)

for count in range(howManyBytes-1, -1, -1):

b[count] = int(n % 256)

n >>= 8

return b

def mpiToNumber(mpi): #mpi is an openssl-format bignum string

if (ord(mpi[4]) & 0x80) !=0: #Make sure this is a positive number

raise AssertionError()

b = bytearray(mpi[4:])

return bytesToNumber(b)

def numberToMPI(n):

b = numberToByteArray(n)

ext = 0

#If the high-order bit is going to be set,

#add an extra byte of zeros

if (numBits(n) & 0x7)==0:

ext = 1

length = numBytes(n) + ext

b = bytearray(4+ext) + b

b[0] = (length >> 24) & 0xFF

b[1] = (length >> 16) & 0xFF

b[2] = (length >> 8) & 0xFF

b[3] = length & 0xFF

return bytes(b)

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Misc. Utility Functions

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

def numBits(n):

if n==0:

return 0

s = "%x" % n

return ((len(s)-1)\*4) + \

{'0':0, '1':1, '2':2, '3':2,

'4':3, '5':3, '6':3, '7':3,

'8':4, '9':4, 'a':4, 'b':4,

'c':4, 'd':4, 'e':4, 'f':4,

}[s[0]]

return int(math.floor(math.log(n, 2))+1)

def numBytes(n):

if n==0:

return 0

bits = numBits(n)

return int(math.ceil(bits / 8.0))

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Big Number Math

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

def getRandomNumber(low, high):

if low >= high:

raise AssertionError()

howManyBits = numBits(high)

howManyBytes = numBytes(high)

lastBits = howManyBits % 8

while 1:

bytes = getRandomBytes(howManyBytes)

if lastBits:

bytes[0] = bytes[0] % (1 << lastBits)

n = bytesToNumber(bytes)

if n >= low and n < high:

return n

def gcd(a,b):

a, b = max(a,b), min(a,b)

while b:

a, b = b, a % b

return a

def lcm(a, b):

return (a \* b) // gcd(a, b)

#Returns inverse of a mod b, zero if none

#Uses Extended Euclidean Algorithm

def invMod(a, b):

c, d = a, b

uc, ud = 1, 0

while c != 0:

q = d // c

c, d = d-(q\*c), c

uc, ud = ud - (q \* uc), uc

if d == 1:

return ud % b

return 0

if gmpyLoaded:

def powMod(base, power, modulus):

base = gmpy.mpz(base)

power = gmpy.mpz(power)

modulus = gmpy.mpz(modulus)

result = pow(base, power, modulus)

return long(result)

else:

def powMod(base, power, modulus):

if power < 0:

result = pow(base, power\*-1, modulus)

result = invMod(result, modulus)

return result

else:

return pow(base, power, modulus)

#Pre-calculate a sieve of the ~100 primes < 1000:

def makeSieve(n):

sieve = list(range(n))

for count in range(2, int(math.sqrt(n))+1):

if sieve[count] == 0:

continue

x = sieve[count] \* 2

while x < len(sieve):

sieve[x] = 0

x += sieve[count]

sieve = [x for x in sieve[2:] if x]

return sieve

sieve = makeSieve(1000)

def isPrime(n, iterations=5, display=False):

#Trial division with sieve

for x in sieve:

if x >= n: return True

if n % x == 0: return False

#Passed trial division, proceed to Rabin-Miller

#Rabin-Miller implemented per Ferguson & Schneier

#Compute s, t for Rabin-Miller

if display: print("\*", end=' ')

s, t = n-1, 0

while s % 2 == 0:

s, t = s//2, t+1

#Repeat Rabin-Miller x times

a = 2 #Use 2 as a base for first iteration speedup, per HAC

for count in range(iterations):

v = powMod(a, s, n)

if v==1:

continue

i = 0

while v != n-1:

if i == t-1:

return False

else:

v, i = powMod(v, 2, n), i+1

a = getRandomNumber(2, n)

return True

def getRandomPrime(bits, display=False):

if bits < 10:

raise AssertionError()

#The 1.5 ensures the 2 MSBs are set

#Thus, when used for p,q in RSA, n will have its MSB set

#

#Since 30 is lcm(2,3,5), we'll set our test numbers to

#29 % 30 and keep them there

low = ((2 \*\* (bits-1)) \* 3) // 2

high = 2 \*\* bits - 30

p = getRandomNumber(low, high)

p += 29 - (p % 30)

while 1:

if display: print(".", end=' ')

p += 30

if p >= high:

p = getRandomNumber(low, high)

p += 29 - (p % 30)

if isPrime(p, display=display):

return p

#Unused at the moment...

def getRandomSafePrime(bits, display=False):

if bits < 10:

raise AssertionError()

#The 1.5 ensures the 2 MSBs are set

#Thus, when used for p,q in RSA, n will have its MSB set

#

#Since 30 is lcm(2,3,5), we'll set our test numbers to

#29 % 30 and keep them there

low = (2 \*\* (bits-2)) \* 3//2

high = (2 \*\* (bits-1)) - 30

q = getRandomNumber(low, high)

q += 29 - (q % 30)

while 1:

if display: print(".", end=' ')

q += 30

if (q >= high):

q = getRandomNumber(low, high)

q += 29 - (q % 30)

#Ideas from Tom Wu's SRP code

#Do trial division on p and q before Rabin-Miller

if isPrime(q, 0, display=display):

p = (2 \* q) + 1

if isPrime(p, display=display):

if isPrime(q, display=display):

return p