# Authors:

# Bram Cohen

# Trevor Perrin - various changes

#

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

# Also see Bram Cohen's statement below

"""

A pure python (slow) implementation of rijndael with a decent interface

To include -

from rijndael import rijndael

To do a key setup -

r = rijndael(key, block\_size = 16)

key must be a string of length 16, 24, or 32

blocksize must be 16, 24, or 32. Default is 16

To use -

ciphertext = r.encrypt(plaintext)

plaintext = r.decrypt(ciphertext)

If any strings are of the wrong length a ValueError is thrown

"""

# ported from the Java reference code by Bram Cohen, bram@gawth.com, April 2001

# this code is public domain, unless someone makes

# an intellectual property claim against the reference

# code, in which case it can be made public domain by

# deleting all the comments and renaming all the variables

import copy

import string

shifts = [[[0, 0], [1, 3], [2, 2], [3, 1]],

[[0, 0], [1, 5], [2, 4], [3, 3]],

[[0, 0], [1, 7], [3, 5], [4, 4]]]

# [keysize][block\_size]

num\_rounds = {16: {16: 10, 24: 12, 32: 14}, 24: {16: 12, 24: 12, 32: 14}, 32: {16: 14, 24: 14, 32: 14}}

A = [[1, 1, 1, 1, 1, 0, 0, 0],

[0, 1, 1, 1, 1, 1, 0, 0],

[0, 0, 1, 1, 1, 1, 1, 0],

[0, 0, 0, 1, 1, 1, 1, 1],

[1, 0, 0, 0, 1, 1, 1, 1],

[1, 1, 0, 0, 0, 1, 1, 1],

[1, 1, 1, 0, 0, 0, 1, 1],

[1, 1, 1, 1, 0, 0, 0, 1]]

# produce log and alog tables, needed for multiplying in the

# field GF(2^m) (generator = 3)

alog = [1]

for i in range(255):

j = (alog[-1] << 1) ^ alog[-1]

if j & 0x100 != 0:

j ^= 0x11B

alog.append(j)

log = [0] \* 256

for i in range(1, 255):

log[alog[i]] = i

# multiply two elements of GF(2^m)

def mul(a, b):

if a == 0 or b == 0:

return 0

return alog[(log[a & 0xFF] + log[b & 0xFF]) % 255]

# substitution box based on F^{-1}(x)

box = [[0] \* 8 for i in range(256)]

box[1][7] = 1

for i in range(2, 256):

j = alog[255 - log[i]]

for t in range(8):

box[i][t] = (j >> (7 - t)) & 0x01

B = [0, 1, 1, 0, 0, 0, 1, 1]

# affine transform: box[i] <- B + A\*box[i]

cox = [[0] \* 8 for i in range(256)]

for i in range(256):

for t in range(8):

cox[i][t] = B[t]

for j in range(8):

cox[i][t] ^= A[t][j] \* box[i][j]

# S-boxes and inverse S-boxes

S = [0] \* 256

Si = [0] \* 256

for i in range(256):

S[i] = cox[i][0] << 7

for t in range(1, 8):

S[i] ^= cox[i][t] << (7-t)

Si[S[i] & 0xFF] = i

# T-boxes

G = [[2, 1, 1, 3],

[3, 2, 1, 1],

[1, 3, 2, 1],

[1, 1, 3, 2]]

AA = [[0] \* 8 for i in range(4)]

for i in range(4):

for j in range(4):

AA[i][j] = G[i][j]

AA[i][i+4] = 1

for i in range(4):

pivot = AA[i][i]

if pivot == 0:

t = i + 1

while AA[t][i] == 0 and t < 4:

t += 1

assert t != 4, 'G matrix must be invertible'

for j in range(8):

AA[i][j], AA[t][j] = AA[t][j], AA[i][j]

pivot = AA[i][i]

for j in range(8):

if AA[i][j] != 0:

AA[i][j] = alog[(255 + log[AA[i][j] & 0xFF] - log[pivot & 0xFF]) % 255]

for t in range(4):

if i != t:

for j in range(i+1, 8):

AA[t][j] ^= mul(AA[i][j], AA[t][i])

AA[t][i] = 0

iG = [[0] \* 4 for i in range(4)]

for i in range(4):

for j in range(4):

iG[i][j] = AA[i][j + 4]

def mul4(a, bs):

if a == 0:

return 0

r = 0

for b in bs:

r <<= 8

if b != 0:

r = r | mul(a, b)

return r

T1 = []

T2 = []

T3 = []

T4 = []

T5 = []

T6 = []

T7 = []

T8 = []

U1 = []

U2 = []

U3 = []

U4 = []

for t in range(256):

s = S[t]

T1.append(mul4(s, G[0]))

T2.append(mul4(s, G[1]))

T3.append(mul4(s, G[2]))

T4.append(mul4(s, G[3]))

s = Si[t]

T5.append(mul4(s, iG[0]))

T6.append(mul4(s, iG[1]))

T7.append(mul4(s, iG[2]))

T8.append(mul4(s, iG[3]))

U1.append(mul4(t, iG[0]))

U2.append(mul4(t, iG[1]))

U3.append(mul4(t, iG[2]))

U4.append(mul4(t, iG[3]))

# round constants

rcon = [1]

r = 1

for t in range(1, 30):

r = mul(2, r)

rcon.append(r)

del A

del AA

del pivot

del B

del G

del box

del log

del alog

del i

del j

del r

del s

del t

del mul

del mul4

del cox

del iG

class rijndael:

def \_\_init\_\_(self, key, block\_size = 16):

if block\_size != 16 and block\_size != 24 and block\_size != 32:

raise ValueError('Invalid block size: ' + str(block\_size))

if len(key) != 16 and len(key) != 24 and len(key) != 32:

raise ValueError('Invalid key size: ' + str(len(key)))

self.block\_size = block\_size

ROUNDS = num\_rounds[len(key)][block\_size]

BC = block\_size // 4

# encryption round keys

Ke = [[0] \* BC for i in range(ROUNDS + 1)]

# decryption round keys

Kd = [[0] \* BC for i in range(ROUNDS + 1)]

ROUND\_KEY\_COUNT = (ROUNDS + 1) \* BC

KC = len(key) // 4

# copy user material bytes into temporary ints

tk = []

for i in range(0, KC):

tk.append((key[i \* 4] << 24) | (key[i \* 4 + 1] << 16) |

(key[i \* 4 + 2] << 8) | key[i \* 4 + 3])

# copy values into round key arrays

t = 0

j = 0

while j < KC and t < ROUND\_KEY\_COUNT:

Ke[t // BC][t % BC] = tk[j]

Kd[ROUNDS - (t // BC)][t % BC] = tk[j]

j += 1

t += 1

tt = 0

rconpointer = 0

while t < ROUND\_KEY\_COUNT:

# extrapolate using phi (the round key evolution function)

tt = tk[KC - 1]

tk[0] ^= (S[(tt >> 16) & 0xFF] & 0xFF) << 24 ^ \

(S[(tt >> 8) & 0xFF] & 0xFF) << 16 ^ \

(S[ tt & 0xFF] & 0xFF) << 8 ^ \

(S[(tt >> 24) & 0xFF] & 0xFF) ^ \

(rcon[rconpointer] & 0xFF) << 24

rconpointer += 1

if KC != 8:

for i in range(1, KC):

tk[i] ^= tk[i-1]

else:

for i in range(1, KC // 2):

tk[i] ^= tk[i-1]

tt = tk[KC // 2 - 1]

tk[KC // 2] ^= (S[ tt & 0xFF] & 0xFF) ^ \

(S[(tt >> 8) & 0xFF] & 0xFF) << 8 ^ \

(S[(tt >> 16) & 0xFF] & 0xFF) << 16 ^ \

(S[(tt >> 24) & 0xFF] & 0xFF) << 24

for i in range(KC // 2 + 1, KC):

tk[i] ^= tk[i-1]

# copy values into round key arrays

j = 0

while j < KC and t < ROUND\_KEY\_COUNT:

Ke[t // BC][t % BC] = tk[j]

Kd[ROUNDS - (t // BC)][t % BC] = tk[j]

j += 1

t += 1

# inverse MixColumn where needed

for r in range(1, ROUNDS):

for j in range(BC):

tt = Kd[r][j]

Kd[r][j] = U1[(tt >> 24) & 0xFF] ^ \

U2[(tt >> 16) & 0xFF] ^ \

U3[(tt >> 8) & 0xFF] ^ \

U4[ tt & 0xFF]

self.Ke = Ke

self.Kd = Kd

def encrypt(self, plaintext):

if len(plaintext) != self.block\_size:

raise ValueError('wrong block length, expected ' + str(self.block\_size) + ' got ' + str(len(plaintext)))

Ke = self.Ke

BC = self.block\_size // 4

ROUNDS = len(Ke) - 1

if BC == 4:

SC = 0

elif BC == 6:

SC = 1

else:

SC = 2

s1 = shifts[SC][1][0]

s2 = shifts[SC][2][0]

s3 = shifts[SC][3][0]

a = [0] \* BC

# temporary work array

t = []

# plaintext to ints + key

for i in range(BC):

t.append((plaintext[i \* 4 ] << 24 |

plaintext[i \* 4 + 1] << 16 |

plaintext[i \* 4 + 2] << 8 |

plaintext[i \* 4 + 3] ) ^ Ke[0][i])

# apply round transforms

for r in range(1, ROUNDS):

for i in range(BC):

a[i] = (T1[(t[ i ] >> 24) & 0xFF] ^

T2[(t[(i + s1) % BC] >> 16) & 0xFF] ^

T3[(t[(i + s2) % BC] >> 8) & 0xFF] ^

T4[ t[(i + s3) % BC] & 0xFF] ) ^ Ke[r][i]

t = copy.copy(a)

# last round is special

result = []

for i in range(BC):

tt = Ke[ROUNDS][i]

result.append((S[(t[ i ] >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)

result.append((S[(t[(i + s1) % BC] >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)

result.append((S[(t[(i + s2) % BC] >> 8) & 0xFF] ^ (tt >> 8)) & 0xFF)

result.append((S[ t[(i + s3) % BC] & 0xFF] ^ tt ) & 0xFF)

return bytearray(result)

def decrypt(self, ciphertext):

if len(ciphertext) != self.block\_size:

raise ValueError('wrong block length, expected ' + str(self.block\_size) + ' got ' + str(len(plaintext)))

Kd = self.Kd

BC = self.block\_size // 4

ROUNDS = len(Kd) - 1

if BC == 4:

SC = 0

elif BC == 6:

SC = 1

else:

SC = 2

s1 = shifts[SC][1][1]

s2 = shifts[SC][2][1]

s3 = shifts[SC][3][1]

a = [0] \* BC

# temporary work array

t = [0] \* BC

# ciphertext to ints + key

for i in range(BC):

t[i] = (ciphertext[i \* 4 ] << 24 |

ciphertext[i \* 4 + 1] << 16 |

ciphertext[i \* 4 + 2] << 8 |

ciphertext[i \* 4 + 3] ) ^ Kd[0][i]

# apply round transforms

for r in range(1, ROUNDS):

for i in range(BC):

a[i] = (T5[(t[ i ] >> 24) & 0xFF] ^

T6[(t[(i + s1) % BC] >> 16) & 0xFF] ^

T7[(t[(i + s2) % BC] >> 8) & 0xFF] ^

T8[ t[(i + s3) % BC] & 0xFF] ) ^ Kd[r][i]

t = copy.copy(a)

# last round is special

result = []

for i in range(BC):

tt = Kd[ROUNDS][i]

result.append((Si[(t[ i ] >> 24) & 0xFF] ^ (tt >> 24)) & 0xFF)

result.append((Si[(t[(i + s1) % BC] >> 16) & 0xFF] ^ (tt >> 16)) & 0xFF)

result.append((Si[(t[(i + s2) % BC] >> 8) & 0xFF] ^ (tt >> 8)) & 0xFF)

result.append((Si[ t[(i + s3) % BC] & 0xFF] ^ tt ) & 0xFF)

return bytearray(result)

def encrypt(key, block):

return rijndael(key, len(block)).encrypt(block)

def decrypt(key, block):

return rijndael(key, len(block)).decrypt(block)

def test():

def t(kl, bl):

b = 'b' \* bl

r = rijndael('a' \* kl, bl)

assert r.decrypt(r.encrypt(b)) == b

t(16, 16)

t(16, 24)

t(16, 32)

t(24, 16)

t(24, 24)

t(24, 32)

t(32, 16)

t(32, 24)

t(32, 32)