ASCON:

```
debug = False

debugpermutation = False

# === Ascon AEAD encryption and decryption ===

def ascon_encrypt(key, nonce, associateddata, plaintext, variant="Ascon-
128"):

    assert(len(nonce) == 16 and (len(key) == 16 or (len(key) == 20 and
variant == "Ascon-80pq")))

    S = [0, 0, 0, 0, 0]

    k = len(key) * 8   # bits

    a = 12   # rounds

    b = 8 if variant == "Ascon-128a" else 6   # rounds

    rate = 16 if variant == "Ascon-128a" else 8   # bytes

    ascon_initialize(S, k, rate, a, b, key, nonce)

    ascon_process_associated_data(S, b, rate, associateddata)

    ciphertext = ascon_process_plaintext(S, b, rate, plaintext)

    tag = ascon_finalize(S, rate, a, key)

    return ciphertext + tag

def ascon_decrypt(key, nonce, associateddata, ciphertext, variant="Ascon-
128"):

    assert(len(nonce) == 16 and (len(key) == 16 or (len(key) == 20 and
variant == "Ascon-80pq")))

    assert(len(ciphertext) >= 16)

    S = [0, 0, 0, 0, 0]

    k = len(key) * 8 # bits

    a = 12 # rounds

    b = 8 if variant == "Ascon-128a" else 6   # rounds

    rate = 16 if variant == "Ascon-128a" else 8   # bytes

    ascon_initialize(S, k, rate, a, b, key, nonce)

    ascon_process_associated_data(S, b, rate, associateddata)

    plaintext = ascon_process_ciphertext(S, b, rate, ciphertext[:-16])

    tag = ascon_finalize(S, rate, a, key)

    if tag == ciphertext[-16:]:
```

```
        return plaintext
    else:
        return None
# === Ascon AEAD building blocks ===
def ascon_initialize(S, k, rate, a, b, key, nonce):
    iv_zero_key_nonce = to_bytes([k, rate * 8, a, b] + (20-len(key))*[0]) +
key + nonce
    S[0], S[1], S[2], S[3], S[4] = bytes_to_state(iv_zero_key_nonce)
    if debug: printstate(S, "initial value:")
    ascon_permutation(S, a)
    zero_key = bytes_to_state(zero_bytes(40-len(key)) + key)
    S[0] ^= zero_key[0]
    S[1] ^= zero_key[1]
    S[2] ^= zero_key[2]
    S[3] ^= zero_key[3]
    S[4] ^= zero_key[4]
    if debug: printstate(S, "initialization:")
def ascon_process_associated_data(S, b, rate, associateddata):
    if len(associateddata) > 0:
        a_zeros = rate - (len(associateddata) % rate) - 1
        a_padding = to_bytes([0x80] + [0 for i in range(a_zeros)])
        a_padded = associateddata + a_padding
        for block in range(0, len(a_padded), rate):
            S[0] ^= bytes_to_int(a_padded[block:block+8])
            if rate == 16:
                S[1] ^= bytes_to_int(a_padded[block+8:block+16])
            ascon_permutation(S, b)
    S[4] ^= 1
    if debug: printstate(S, "process associated data:")
def ascon_process_plaintext(S, b, rate, plaintext):
    p_lastlen = len(plaintext) % rate
    p_padding = to_bytes([0x80] + (rate-p_lastlen-1)*[0x00])
```

```
    p_padded = plaintext + p_padding

    # first t-1 blocks

    ciphertext = to_bytes([])

    for block in range(0, len(p_padded) - rate, rate):

        if rate == 8:

            S[0] ^= bytes_to_int(p_padded[block:block+8])

            ciphertext += int_to_bytes(S[0], 8)

        elif rate == 16:

            S[0] ^= bytes_to_int(p_padded[block:block+8])

            S[1] ^= bytes_to_int(p_padded[block+8:block+16])

            ciphertext += (int_to_bytes(S[0], 8) + int_to_bytes(S[1], 8))

        ascon_permutation(S, b)

    # last block t

    block = len(p_padded) - rate

    if rate == 8:

        S[0] ^= bytes_to_int(p_padded[block:block+8])

        ciphertext += int_to_bytes(S[0], 8)[:p_lastlen]

    elif rate == 16:

        S[0] ^= bytes_to_int(p_padded[block:block+8])

        S[1] ^= bytes_to_int(p_padded[block+8:block+16])

        ciphertext += (int_to_bytes(S[0], 8)[:min(8,p_lastlen)] +
int_to_bytes(S[1], 8)[:max(0,p_lastlen-8)])

    if debug: printstate(S, "process plaintext:")

    return ciphertext

def ascon_process_ciphertext(S, b, rate, ciphertext):

    c_lastlen = len(ciphertext) % rate

    c_padded = ciphertext + zero_bytes(rate - c_lastlen)

    # first t-1 blocks

    plaintext = to_bytes([])

    for block in range(0, len(c_padded) - rate, rate):

        if rate == 8:

            Ci = bytes_to_int(c_padded[block:block+8])
```

```
            plaintext += int_to_bytes(S[0] ^ Ci, 8)
            S[0] = Ci
        elif rate == 16:
            Ci = (bytes_to_int(c_padded[block:block+8]), 
bytes_to_int(c_padded[block+8:block+16]))
            plaintext += (int_to_bytes(S[0] ^ Ci[0], 8) + int_to_bytes(S[1] ^ 
Ci[1], 8))
            S[0] = Ci[0]
            S[1] = Ci[1]
        ascon_permutation(S, b)
    # last block t
    block = len(c_padded) - rate
    if rate == 8:
        c_padding1 = (0x80 << (rate-c_lastlen-1)*8)
        c_mask = (0xFFFFFFFFFFFFFFFF >> (c_lastlen*8))
        Ci = bytes_to_int(c_padded[block:block+8])
        plaintext += int_to_bytes(Ci ^ S[0], 8)[:c_lastlen]
        S[0] = Ci ^ (S[0] & c_mask) ^ c_padding1
    elif rate == 16:
        c_lastlen_word = c_lastlen % 8
        c_padding1 = (0x80 << (8-c_lastlen_word-1)*8)
        c_mask = (0xFFFFFFFFFFFFFFFF >> (c_lastlen_word*8))
        Ci = (bytes_to_int(c_padded[block:block+8]), 
bytes_to_int(c_padded[block+8:block+16]))
        plaintext += (int_to_bytes(S[0] ^ Ci[0], 8) + int_to_bytes(S[1] ^ 
Ci[1], 8))[:c_lastlen]
        if c_lastlen < 8:
            S[0] = Ci[0] ^ (S[0] & c_mask) ^ c_padding1
        else:
            S[0] = Ci[0]
            S[1] = Ci[1] ^ (S[1] & c_mask) ^ c_padding1
    if debug: printstate(S, "process ciphertext:")
    return plaintext
def ascon_finalize(S, rate, a, key):
```

```
    assert(len(key) in [16,20])
    S[rate//8+0] ^= bytes_to_int(key[0:8])
    S[rate//8+1] ^= bytes_to_int(key[8:16])
    S[rate//8+2] ^= bytes_to_int(key[16:])
    ascon_permutation(S, a)
    S[3] ^= bytes_to_int(key[-16:-8])
    S[4] ^= bytes_to_int(key[-8:])
    tag = int_to_bytes(S[3], 8) + int_to_bytes(S[4], 8)
    if debug: printstate(S, "finalization:")
    return tag
# === Ascon permutation ===
def ascon_permutation(S, rounds=1):
    assert(rounds <= 12)
    if debugpermutation: printwords(S, "permutation input:")
    for r in range(12-rounds, 12):
        # --- add round constants ---
        S[2] ^= (0xf0 - r*0x10 + r*0x1)
        if debugpermutation: printwords(S, "round constant addition:")
        # --- substitution layer ---
        S[0] ^= S[4]
        S[4] ^= S[3]
        S[2] ^= S[1]
        T = [(S[i] ^ 0xFFFFFFFFFFFFFFFF) & S[(i+1)%5] for i in range(5)]
        for i in range(5):
            S[i] ^= T[(i+1)%5]
        S[1] ^= S[0]
        S[0] ^= S[4]
        S[3] ^= S[2]
        S[2] ^= 0XFFFFFFFFFFFFFFFF
        if debugpermutation: printwords(S, "substitution layer:")
        # --- linear diffusion layer ---
        S[0] ^= rotr(S[0], 19) ^ rotr(S[0], 28)
```

```
        S[1] ^= rotr(S[1], 61) ^ rotr(S[1], 39)
        S[2] ^= rotr(S[2],  1) ^ rotr(S[2],  6)
        S[3] ^= rotr(S[3], 10) ^ rotr(S[3], 17)
        S[4] ^= rotr(S[4],  7) ^ rotr(S[4], 41)
        if debugpermutation: printwords(S, "linear diffusion layer:")
# === helper functions ===
def get_random_bytes(num):
    import os
    return to_bytes(os.urandom(num))
def zero_bytes(n):
    return n * b"\x00"
def to_bytes(l): # where l is a list or bytearray or bytes
    return bytes(bytearray(l))
def bytes_to_int(bytes):
    return sum([bi << ((len(bytes) - 1 - i)*8) for i, bi in 
enumerate(to_bytes(bytes))])
def bytes_to_state(bytes):
    return [bytes_to_int(bytes[8*w:8*(w+1)]) for w in range(5)]
def int_to_bytes(integer, nbytes):
    return to_bytes([(integer >> ((nbytes - 1 - i) * 8)) % 256 for i in 
range(nbytes)])


def rotr(val, r):
    return ((val >> r) ^ (val << (64-r))) % (1 << 64)
def bytes_to_hex(b):
    return b.hex()
    #return "".join(x.encode('hex') for x in b)
def printstate(S, description=""):
    print(" " + description)
    print(" ".join(["{s:016x}".format(s=s) for s in S]))
def printwords(S, description=""):
    print(" " + description)
```

```
    print("\n".join(["  x{i}={s:016x}".format(**locals()) for i, s in 
enumerate(S)]))

# === some demo if called directly ===

def demo_print(data):

    maxlen = max([len(text) for (text, val) in data])

    for text, val in data:

        print("{text}:{align} 0x{val} ({length} bytes)".format(text=text, 
align=((maxlen - len(text)) * " "), val=bytes_to_hex(val), length=len(val)))

def demo_aead(variant):

    assert variant in ["Ascon-128", "Ascon-128a", "Ascon-80pq"]

    keysize = 20 if variant == "Ascon-80pq" else 16

    print("=== demo encryption using {variant} ===".format(variant=variant))

    key   = zero_bytes(keysize) # get_random_bytes(keysize)

    nonce = zero_bytes(16)      # get_random_bytes(16)

    associateddata = b"ASCON"

    plaintext      = b"ascon"

    ciphertext        = ascon_encrypt(key, nonce, associateddata, plaintext, 
variant)

    receivedplaintext = ascon_decrypt(key, nonce, associateddata, ciphertext, 
variant)

   if receivedplaintext == None: print("verification failed!")

    demo_print([("key", key),

                ("nonce", nonce),

                ("plaintext", plaintext),

                ("ass.data", associateddata),

                ("ciphertext", ciphertext[:-16]),

                ("tag", ciphertext[-16:]),

                ("received", receivedplaintext),

               ])

if __name__ == "__main__":

    demo_aead("Ascon-128")
```

```python
    demo_print([("key", key),
                ("nonce", nonce),
                ("plaintext", plaintext),
                ("ass.data", associateddata),
                ("ciphertext", ciphertext[:-16]),
                ("tag", ciphertext[-16:]),
                ("received", receivedplaintext),
               ])

def demo_hash(variant="Ascon-Hash", hashlength=32):
    assert variant in ["Ascon-Xof", "Ascon-Hash"]
    print("=== demo hash using {variant} ===".format(variant=variant))

    message = b"ascon"
    tag = ascon_hash(message, variant, hashlength)

    demo_print([("message", message), ("tag", tag)])


if __name__ == '__main__':
    demo_aead("Ascon-128")
    #demo_hash("Ascon-Hash")
```

```
=== demo encryption using Ascon-128 ===
key:        0x00000000000000000000000000000000 (16 bytes)
nonce:      0x00000000000000000000000000000000 (16 bytes)
plaintext:  0x6173636f6e (5 bytes)
ass.data:   0x4153434f4e (5 bytes)
ciphertext: 0x868862140e (5 bytes)
tag:        0xad65f5942258dad53caa7a56f3a292d8 (16 bytes)
received:   0x6173636f6e (5 bytes)
```

PRESENT:

```c
#include <stdio.h>
#include<stdint.h>

typedef struct __attribute__((__packed__)) byte{
    uint8_t nibble1 : 4;
    uint8_t nibble2 : 4;
} byte;
uint8_t S[] = {0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD, 0x3, 0xE, 0xF,
0x8, 0x4, 0x7, 0x1, 0x2};
uint8_t invS[] = {0x5, 0xe, 0xf, 0x8, 0xC, 0x1, 0x2, 0xD, 0xB, 0x4,
0x6, 0x3, 0x0, 0x7, 0x9, 0xA};
uint8_t P[] = {0, 16, 32, 48, 1, 17, 33, 49, 2, 18, 34, 50, 3, 19, 35,
51,
                    4, 20, 36, 52, 5, 21, 37, 53, 6, 22, 38, 54, 7,
23, 39, 55,
```

```
                    8, 24, 40, 56, 9, 25, 41, 57, 10, 26, 42, 58, 11,
27, 43, 59,
                    12, 28, 44, 60, 13, 29, 45, 61, 14, 30, 46, 62,
15, 31, 47, 63};
byte* fromHexStringToBytes (char *block){
    byte* bytes = malloc(8 * sizeof(byte));
    int i;
    for (i=0; i<8; i++){
        bytes[i].nibble1 = (block[2*i]>='0' && block[2*i]<='9')?
(block[2*i] - '0') : (block[2*i] - 'a' + 10);
        bytes[i].nibble2 = (block[2*i+1]>='0' && block[2*i+1]<='9')?
(block[2*i+1] - '0') : (block[2*i+1] - 'a' + 10);
    }
    return bytes;
}
// function for converting an array of bytes to a 64-bit integer
uint64_t fromBytesToLong (byte* bytes){
    uint64_t result = 0;
    int i;
    for (i=0; i<8; i++){
        result = (result << 4) | (bytes[i].nibble1 & 0xFUL);
        result = (result << 4) | (bytes[i].nibble2 & 0xFUL);
    }
    return result;
}
// function for converting Hex String to a 64-bit integer
uint64_t fromHexStringToLong (char* block){
    uint64_t result;
    int i;
    for (i=0; i<16; i++)
        result = (result << 4) | ((block[i]>='0' && block[i]<='9')?
(block[i] - '0') : (block[i] - 'a' + 10));
    return result;
}
// function for converting a 64-bit integer to an array of bytes
byte* fromLongToBytes (uint64_t block){
    byte* bytes = malloc (8 * sizeof(byte));
    int i;
    for (i=7; i>=0; i--){
        bytes[i].nibble2 = (block >> 2 * (7 - i) * 4) & 0xFLL;
        bytes[i].nibble1 = (block >> (2 * (7 - i) + 1) * 4) & 0xFLL;
    }
    return bytes;
}
// function for converting a 64-bit integer to a Hex String
char* fromLongToHexString (uint64_t block){
    char* hexString = malloc (17 * sizeof(char));
    sprintf(hexString, "%016llx", block);
    return hexString;
}
// function for converting a nibble using the SBox
uint8_t Sbox(uint8_t input){
```

```
    return S[input];
}
// inverse function of the one above (used to obtain the original
nibble)
uint8_t inverseSbox(uint8_t input){
    return invS[input];
}

uint64_t permute(uint64_t source){
    uint64_t permutation = 0;
    int i;
    for (i=0; i<64; i++){
        int distance = 63 - i;
        permutation = permutation | ((source >> distance & 0x1) << 63
- P[i]);
    }
    return permutation;
}

uint64_t inversepermute(uint64_t source){
    uint64_t permutation = 0;
    int i;
    for (i=0; i<64; i++){
        int distance = 63 - P[i];
        permutation = (permutation << 1) | ((source >> distance) &
0x1);
    }
    return permutation;
}
// function that returns the low 16 bits of the key, which is given as
input in a Hex String format
uint16_t getKeyLow(char* key){
    int i;
    uint16_t keyLow = 0;
    for (i=16; i<20; i++)
        keyLow = (keyLow << 4) | (((key[i]>='0' && key[i]<='9')?
(key[i] - '0') : (key[i] - 'a' + 10)) & 0xF);
    return keyLow;
}
// function that generates subKeys from the key according to the
PRESENT key scheduling algorithm for a 80-bit key
uint64_t* generateSubkeys(char* key){
    uint64_t keyHigh = fromHexStringToLong(key);
    uint16_t keyLow = getKeyLow(key);
    uint64_t* subKeys = malloc(32 * (sizeof(uint64_t)));
    int i;
    subKeys[0] = keyHigh;
    for (i=1; i<32; i++){
        uint64_t temp1 = keyHigh, temp2 = keyLow;
        keyHigh = (keyHigh << 61) | (temp2 << 45) | (temp1 >> 19);
        keyLow = ((temp1 >> 3) & 0xFFFF);
        uint8_t temp = Sbox(keyHigh >> 60);
```

```
        keyHigh = keyHigh & 0x0FFFFFFFFFFFFFFFLL;
        keyHigh = keyHigh | (((uint64_t)temp) << 60);
        keyLow = keyLow ^ ((i & 0x01) << 15); //k15 is the most
significant bit in keyLow
        keyHigh = keyHigh ^ (i >> 1); //the other bits are the least
significant ones in keyHigh
        subKeys[i] = keyHigh;
    }
    return subKeys;
}
// function for encrypting a block using a key
char* encrypt(char* plaintext, char* key){
    uint64_t* subkeys = generateSubkeys(key);
    uint64_t state = fromHexStringToLong(plaintext);
    int i, j;
    for (i=0; i<31; i++){
        state = state ^ subkeys[i];
        byte* stateBytes = fromLongToBytes(state);
        for (j=0; j<8; j++){
            stateBytes[j].nibble1 = Sbox(stateBytes[j].nibble1);
            stateBytes[j].nibble2 = Sbox(stateBytes[j].nibble2);
        }
        state = permute(fromBytesToLong(stateBytes));
        free(stateBytes);
    }
    //the last round only XORs the state with the round key
    state = state ^ subkeys[31];
    //free the memory of the subkeys (they are not needed anymore)
    free(subkeys);
    return fromLongToHexString(state);
}
// function for decrypting a block using a key
char* decrypt(char* ciphertext, char* key){
    uint64_t* subkeys = generateSubkeys(key);
    uint64_t state = fromHexStringToLong(ciphertext);
    int i, j;
    //apply first 31 rounds
    for (i=0; i<31; i++){
        state = state ^ subkeys[31 - i];
        state = inversepermute(state);
        byte* stateBytes = fromLongToBytes(state);
        for (j=0; j<8; j++){
            stateBytes[j].nibble1 =
inverseSbox(stateBytes[j].nibble1);
            stateBytes[j].nibble2 =
inverseSbox(stateBytes[j].nibble2);
        }
        state = fromBytesToLong(stateBytes);
        free(stateBytes);
    }
    state = state ^ subkeys[0];
    //free the memory of the subkeys (they are not needed anymore)
```

```c
    free(subkeys);
    return fromLongToHexString(state);
}
// Test main function
int main(){
    char *plaintext = malloc(17 * sizeof(char));
    char *key = malloc(21 * sizeof(char));
    char *ciphertext;
    printf("Enter the plaintext (64 bits) in hexadecimal format\nUse
lower case characters and enter new line at the end\n");
    gets(plaintext);
    printf("Enter the key (80 bits) in hexadecimal format\nUse lower
case characters and enter new line at the end\n");
    gets(key);
    ciphertext = encrypt(plaintext, key);
    printf("The ciphertext is: ");
    puts(ciphertext);
    printf("The decrypted plaintext is: ");
    puts(decrypt(ciphertext, key));
    free(key);
    free(plaintext);
    free(ciphertext);
    return 0;
}
```