Cryptography: TD

Sylvain Ruhault

These exercises require that you run `python` and that you have access to a Linux shell.

1 Prerequisites

For each of the following, tell if it refers to an encryption algorithm, a signature algorithm, a key exchange algorithm, a mode of operation, a hash function:

- AES
- SHA3
- ECB
- RSA
- HMAC
- LZ
- DH
- CHACHA-POLY1305

AES-128 refers to:

- The size of the key (16 bytes)
- The size of the key (128 bits)
- The block size (128 bits)

SHA-3 means:

- It is 3 times faster than SHA-1
- It is 3 times more secure than SHA-1
- It is the third version of SHA

Which of the following, is a 128 bits key in decimal, hexadecimal, base 64, binary format:

- `1101110001000010011000110001001001100010100110011011000110100101100110111111101100001101100100101000100110110100001010001100111`
- `6e213189314cd8d2cdfd86c944da1467`
- `NmUyMTMxODkzMTRjZDhkMmNkZmQ4NmM5NDRkYTE0NjcK`
- `146387430040258906480581650393585030247`

2 ECB, CBC modes illustration

ECB and CBC modes of operations are depicted in Figures 1 and 2, respectively.

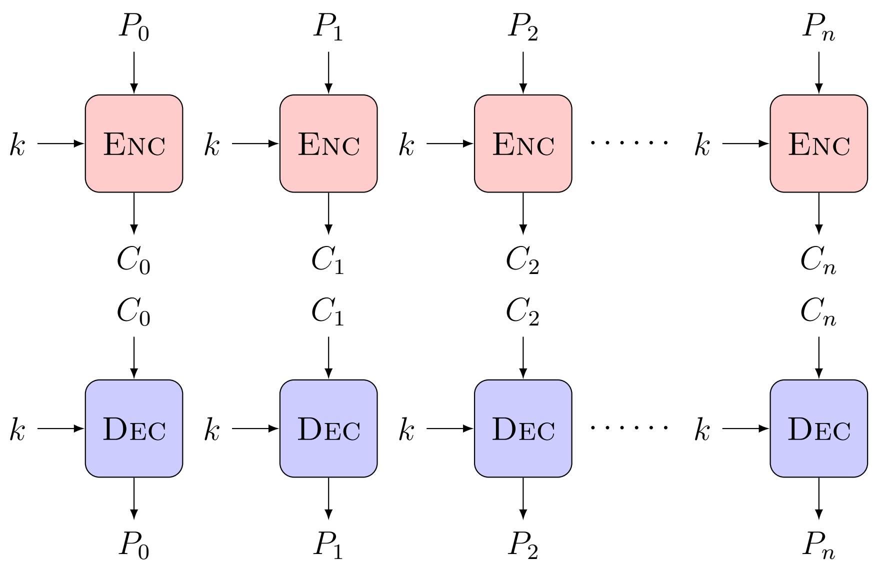

Figure 1: ECB encryption mode

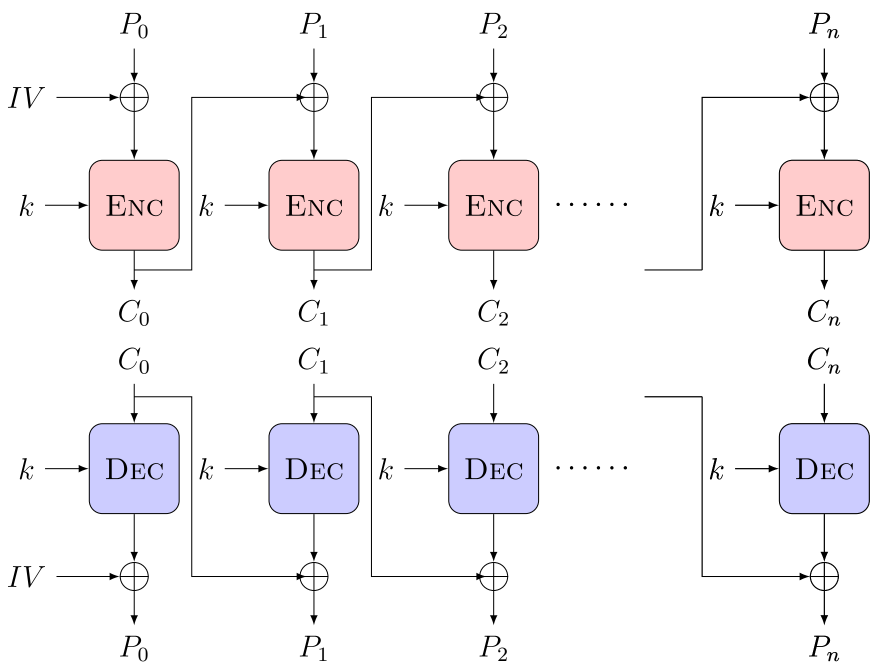

Figure 2: CBC encryption mode

Choose an image, then, assuming "[filename].png" is your file:

- Convert it to "[filename].ppm" format with `convert` function (for Linux) or do it online.
- With the help of `head` function (for Linux and Mac) or `gc` function (for Windows), copy the 3 three first lines of your ppm file to a new file called "[filename].header".

- With the help of `tail` function (for Linux and Mac) or `gc` function (for Windows), copy the content of your ppm file without the 3 first lines in a new file called "[filename].body".
- With the help of `openssl`, encrypt "[filename].body" with AES-128-ECB.
- With the help of `cat` function (for Linux and Mac) or `type` function (for Windows), build a new file with the encrypted content.
- Now open your encrypted image, what do you see ?
- Repeat all previous commands, with AES-128-CBC instead of AES-128-ECB. What do you see ?

3 Key Derivation with `python`

Encryption is a process !

1. Algorithm parameters are selected
2. Key are derived from password
3. Encryption is performed:
   - Authenticated encryption (e.g. AES-GCM):
     - input = (cleartext, key)
     - output = (ciphertext, iv, tag)
   - Not authenticated encryption (e.g. AES-CBC, AES-CTR):
     - input = (cleartext, key)
     - output = (ciphertext, iv)
     - integrity shall be calculated appart !!

3.1 Example 1

In your shell enter the following:

```
pip install pbkdf2
```

Then in a dedicated script, copy the following:

```
import pbkdf2, binascii, os

# Derive a 256-bit AES encryption key from the password
password = "s3cr3tp@ss"
key = pbkdf2.PBKDF2(password).read(32)
print('AES encryption key:', binascii.hexlify(key))
```

Execute the previous script, you should obtain an error:

```
TypeError: PBKDF2.__init__() missing 1 required positional argument: 'salt'
```

What are the parameters of `PBKDF2` function ? Complete the previous code so it works !

3.2 Example 2

In your shell enter the following:

```
pip install backports.pbkdf2
```

Then in a dedicated script, copy the following:

```
import binascii
from backports.pbkdf2 import pbkdf2_hmac

salt = binascii.unhexlify('aaef2d3f4d77ac66e9c5a6c3d8f921d1')
passwd = "s3cr3tp@ss".encode("utf8")
key = pbkdf2_hmac("sha256", passwd, salt)
print("Derived key:", binascii.hexlify(key))
```

Execute the previous script, you should obtain an error:

```
TypeError: pbkdf2_hmac() missing 1 required positional argument: 'iterations'
```

What are the parameters of `pbkdf2_hmac` function ? Complete the previous code so it works !

4 Symmetric Encryption with `python`

4.1 Example 1

In a dedicated script, copy the following:

```
import pyaes, secrets

iv = secrets.randbits(256)
plaintext = "Text for encryption"
aes = pyaes.AESModeOfOperationCTR(key, pyaes.Counter(iv))
ciphertext = aes.encrypt(plaintext)
print('Encrypted:', binascii.hexlify(ciphertext))
```

What is encryption algorithm ? Key size ? Mode of operation ? Execute the previous script, you should obtain an error:

```
NameError: name 'key' is not defined
```

Use one of the two previous key derivation script to update the script to encrypt plaintext with a key derivated from a password. What is the size of the output ?

4.2 Example 2

In your shell enter the following:

```
pip install aes_pkcs5
```

Then in a dedicated script, copy the following:

```
from aes_pkcs5.algorithms.aes_cbc_pkcs5_padding import AESCBCPKCS5Padding
key = "@NcRfUjXn2r5u8x/"
output_format = "hex"
iv = secrets.token_hex(8)
plaintext = "Text for encryption"

cipher = AESCBCPKCS5Padding(key, output_format, iv)
ciphertext = cipher.encrypt(plaintext)
print('Encrypted:', ciphertext)
```

What is encryption algorithm ? Key size ? Mode of operation ? What is the size of the output ?

5 Hash Functions with python

5.1 State of Art

Name	digest size	Pub. in	Developed by	Status
MD2	128	1989	Ronald Rivest	insecure
MD4	128	1990	Ronald Rivest	insecure
SHA-0	160	1993	NSA	insecure
MD5	128	1994	Ronald Rivest	insecure
SHA-1	160	1995	NSA	being deprecated
RIPEMD-160	160	1996	Dobbertin & al.	unknown
SHA-224	224	2004	NIST	secure
SHA-256	256	2001	NIST	secure
SHA-384	384	2001	NIST	secure
SHA-512	512	2001	NIST	secure
KECCAK-224	224	2012	Daemen & al.	secure
KECCAK-256	256	2012	Daemen & al.	secure
KECCAK-384	384	2012	Daemen & al.	secure
KECCAK-512	512	2012	Daemen & al.	secure

5.2 Example 1

In a dedicated script, copy the following:

```
import hashlib, binascii

sha256hash = hashlib.sha256(b'hello').digest()
print("SHA-256: ", binascii.hexlify(sha256hash))

sha3_256 = hashlib.sha3_256(b'hello').digest()
print("SHA3-256:", binascii.hexlify(sha3_256))

blake2s = hashlib.new('blake2s', b'hello').digest()
print("BLAKE2s:   ", binascii.hexlify(blake2s))
```

For each computation, what is the hash function used ? What is the size of the output ?

5.3 Example 2

In a dedicated script, copy the following:

```
import hashlib, binascii

text = 'hello'
data = text.encode("utf8")

sha256hash = hashlib.sha256(data).digest()
print("SHA-256: ", binascii.hexlify(sha256hash))

sha3_256 = hashlib.sha3_256(data).digest()
print("SHA3-256:", binascii.hexlify(sha3_256))

blake2s = hashlib.new('blake2s', data).digest()
print("BLAKE2s:   ", binascii.hexlify(blake2s))
```

Check that you obtain the same results as before !

5.4 Example 3

In your shell enter the following:

```
pip install pycryptodome
```

In a dedicated script, copy the following:

```
from Crypto.Hash import RIPEMD160

ripemd160 = RIPEMD160.new(data=b'hello').digest()
print("RIPEMD-160:", binascii.hexlify(ripemd160))

from Crypto.Hash import keccak

keccak256 = keccak.new(data=b'hello', digest_bits=256).digest()
print("Keccak256:", binascii.hexlify(keccak256))
```

For each computation, what is the hash function used ? What is the size of the output ? Are `keccak256` and `sha3_256` equivalent ?

6 Prerequisites, again

For each of the following, tell if it refers to an encryption algorithm, a signature algorithm, a key exchange algorithm, a mode of operation, a hash function:

- DSA
- Blake2s
- CTR

- ECDH
- `curve25519`

In RSA-2048, 2048 refers to

- The size of the public key
- The size of the private key

7 Algebra

7.1 Content of $(\mathbb{Z}/p\mathbb{Z})^*$, where p is prime

Let p a prime number. We want to implement a `python` function that outputs all content of $(\mathbb{Z}/p\mathbb{Z})^*$. Start with $p = 23$. Choose a generator of the group $(\mathbb{Z}/p\mathbb{Z})^*$: any integer prime with p works! *Hint: use function `pow` from `python`.*

Output of your program shall be similar to:

```
1 2 4 8 16 9 18 13 3 6 12 1 2 4 8 16 9 18 13 3 6 12 1
```

Or to:

```
1 3 9 4 12 13 16 2 6 18 8 1 3 9 4 12 13 16 2 6 18 8 1
```

Now choose a larger prime number and print again the output.

7.2 Basic DH

In a dedicated script, copy the following:

```
from random import randint
q = 509
p = 2*q+1
g = 2
u = randint(2,p-1)
v = randint(2,p-1)
U =
V =
Ka =
Kb =
print(Ka)
print('\n')
print(Kb)
print('\n')
```

Some part of the code is missing, complete it to ensure `Ka = Kb`

7.3 Modular inverse

In a dedicated script, copy the following:

```
import math

def int_to_bytes(n):
    return n.to_bytes((n.bit_length() + 7) // 8, 'big')


def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a


def mod_inv(a, n):
    t, r = 1, a
    new_t, new_r = 0, n

    while new_r != 0:
        quotient = r // new_r
        t, new_t = new_t, t - quotient * new_t
        r, new_r = new_r, r - quotient * new_r
    if r > 1:
        raise Exception("a is not invertible")
    if t < 0:
        t = t + n
    return t
```

What are implemented function ? Let

- $p = 17136853248687850037$
- $q = 10477288835220524183$
- $c = 71849746646825786308004273212656766001$

What is the size of p, q and c in bits ? Complete the file to test the algorithms with p, q, c and the following variables:

- $e = 65537$
- $n = p \times q$
- $d = e^{-1}\ mod\ (p-1) \times (q-1)$
- $m = c^d\ mod\ n$

What is the size of d, n and m in bits ? What is the type of m ? Compute m. What is its size in bits ? Now convert m in bytes !

8 RSA algorithm

8.1 RSA encryption

Key Generation

1. Choose two distinct random prime numbers p and q
2. Compute the **modulus** $n = pq$.
3. Choose e, **the public exponent**, such as
   - $1 < e < \varphi(pq) = (p-1)(q-1)$
   - e and $\varphi(pq)$ are coprime.
4. Find d, **the private exponent**, such as $de \equiv 1\ mod\ \varphi(pq)$

$\Longrightarrow$ **Public key**: (n, e) **Private key**: (n, d)

Encryption of plaintext M into ciphertext C

1. Convert M into an integer m, $0 < m < n$
2. Compute $C = m^e\ mod\ n$

Decryption of ciphertext C into plaintext M

1. Compute $m \equiv C^d\ mod\ n$
2. Convert m into M

8.2 RSA signature

Key Generation

1. Choose two distinct random prime numbers p and q
2. Compute the **modulus** $n = pq$.
3. Choose e, **the public exponent**, such as
   - $1 < e < \varphi(pq) = (p-1)(q-1)$
   - e and $\varphi(pq)$ are coprime.
4. Find d, **the private exponent**, such as $de \equiv 1\ mod\ \varphi(pq)$

$\Longrightarrow$ **Public key**: (n, e) **Private key**: (n, d)

Signature of plaintext M

1. Compute $h = H(M)$
2. Compute $s = h^d\ mod\ n$

Verification of signature s of message M

1. Compute $h = H(M)$
2. Compute $h' = s^e\ mod\ n$
3. Assess $h = h'$

8.3 RSA encryption and signature implementations

In a dedicated script, copy the following:

```
def RSA_generate_keys("""TBC"""):

    return """TBC"""

def RSA_private_exponent("""TBC"""):

    return """TBC"""

def RSA_encrypt("""TBC"""):

    return """TBC"""

def RSA_decrypt("""TBC"""):

    return """TBC"""

def RSA_sign("""TBC"""):

    return """TBC"""

def RSA_verif("""TBC"""):

    return """TBC"""
```

These are prototypes for the following algorithms:

- RSA key generation
- RSA private exponent generation
- RSA encryption
- RSA decryption
- RSA signature generation
- RSA signature verification

For each algorithm, what are the inputs ? Complete these implementations with algorithms `int_to_bytes` and `mod_inv` and use the following import:

```
from Crypto.Util.number import getPrime
```

8.3.1 Encryption implementation test

Let e the public exponent, p, q two prime numbers and a ciphertext c defined with:

$$
\begin{aligned}
e =\ & 65537 \\
p =\ & 13875225427940730164159929091761963230136068001228982029979867389492462526 \\
& 93201513072105635989198737386471226417758902524810453364074473353805279501 \\
& 42805010686258154271615273397398258602252136628038455082064763522545137911 \\
& 98927065738657936397793783490985414375086729903267733205111768929297941132 \\
& 7559100838257 \\
q =\ & 14419139847985888021086043781764150193280541983226719472663275856762836704 \\
& 53679804391872476856237913460472579595648129509197830218658562447373678278 \\
& 27382979815624619802123835380111267878397074533274340907552420874884643793 \\
& 14216565715706951066390591531558186981923397347905595573097898546305662202 \\
& 5886518153717 \\
c =\ & 14648329214499371556114344386966750343752215033479943321915540827554419194 \\
& 99955570243063708596923039742679887047488769058616491084183747906071229886 \\
& 02311118591263697418007432321201599069198114272290985132832399682965920018 \\
& 00010301402827800810717168128401523377682792148456840350944229637594319650 \\
& 10011002600892054279048493506246843366673651239419210732804467402678546888 \\
& 18679343850972884984318855013686257387594067239363933020740106598695521581 \\
& 57387350320814975974924443391962303803194675304477298724100712538627348270 \\
& 97065829958214294800104949896019283996907278769276039934145819769125721018 \\
& 1387234737478353503522259
\end{aligned}
$$

What is the size of p, q and c in bits ? Find the original plaintext !

8.3.2 Signature implementation test

Use (p, q, e) as above to generate a signature s of the original plaintext. What is the size of s in bits ? Check your signature verification implementation with s and with s^* defined with:

$$
\begin{aligned}
s^* =\ & 53002695142922394401575714890521956915604147004529558483355971893111068913 \\
& 16551482249724972958985013063501559498003755295697271926534573005619693254 \\
& 62703676443981776900113156570323266507277564729902622641579209270804433610 \\
& 75439729326835725165706458719994271230498737825995120869421566820875374508 \\
& 74406639306123352918572789358809751856512826156056973669155286922651536821 \\
& 26533925691446216691666838062518204438950436461754679752264505672451110597 \\
& 79155131147022395286994279460082828602209232593878216241767693607999337982 \\
& 03749885544444430125804705885254106537926409896625890871494401008102480196 \\
& 3388531895506657638543 45
\end{aligned}
$$

8.4 Use directly python and openssl

In a dedicated script, copy the following:

```
from Crypto.PublicKey import RSA
key = RSA.generate(2048)
private_key = key.export_key()
file_out = open("private.pem", "wb")
file_out.write(private_key)
file_out.close()
public_key = key.publickey().export_key()
file_out = open("public.pem", "wb")
```

```
file_out.write(public_key)
file_out.close()
```

What is the size of the RSA key ? With the help of `cat` function (for Linux and Mac) or `type` function (for Windows), print the content of files `private.pem` and `public.pem`. Now with `openssl`, decode the files to get all encoded components. *Hint: use `openssl rsa`.*

9 Attacks on RSA algorithm

9.1 Close primes attack

Let $n = p \times q$ an RSA product. An usual recommandation is to generate primes that are *not too close*. Suppose that on the opposite, someone did *not* follow this recommandation, and used the following algorithm to generate p and q:

```
# This is not a secure RSA prime numbers generation. For education only !
from Crypto.Util.number import getPrime
p = getPrime(1024)
q = nextprime(p + 2**519)
n = p*q
```

What is the difference between p and q ? Using the product equality $a^2 - b^2 = (a + b) \times (a - b)$, construct an efficient algorithm to recover p and q from n. You can start with small size n, e.g. $n = 12319 = 127 \times 97$ (hint: $\sqrt{n} \approx 110.9$).

Implement your algorithm. Hint: you can use the following import in `python`:

```
from sympy import sqrt, log, ceiling, Integer
```

Let e a public exponent, n a RSA modulus generated with the previous algorithm and a ciphertext c encrypted defined with:

$$
\begin{aligned}
e = {} & 65537 \\
n = {} & 23668362559912334821487569511015860048734938117176197612535460050680404719 \\
& 68232961405652308151202261170513420302901641793054647906102160567968518368 \\
& 25624909678219412061263841278642162143254598153793839108490167274991216158 \\
& 93120456633221447779176207091549946437332544875687741088425223620429946961 \\
& 50552766386163938164499371677546824381467508290137526419020459763796798317 \\
& 15996972917052991523576778487567599836747201482493257749911654229666717173 \\
& 86238464284133059205909067808167454751995275291932884405650162798067565217 \\
& 83403633094316522088014595280658751068879852228504290550686110897052965574 \\
& 9304772869059017659304881 \\
c = {} & 92204238119193834280803558178821576607537315608633788272536537641761923450 \\
& 72440397188637244477814151233308055894796092737522053881962537973896977854 \\
& 85720925474502117607815188112000126012810885057610177141005118133791618804 \\
& 03733348994036394931052379454438861840569459286903512161300125687899860268 \\
& 47420914470112331736162021504947349288617132583785624228600020473986635225 \\
& 85272344200712181651791173415275531887516568131975347088237835915639319725 \\
& 00140779303371563773079074063119066094310746503715155987214751102323438080 \\
& 76481590763194135384978173591794232844792526103136794027156288462237679399 \\
& 4679298353787839154562033
\end{aligned}
$$

What is the size of n and c in bits ? Use your previous algorithm and your RSA implementation to find the original plaintext !

9.2 Cube root attack

Consider RSA encryption with public exponent $e = 3$. What is encryption algorithm in this case ? Suppose now that someone encrypts the same message m with 3 different public keys n_1, n_2, n_3 (i.e the same message is sent to 3 different people). You have access to the 3 corresponding ciphertexts c_1, c_2, c_3. Write down encryption equations linking $m, n_1, n_2, n_3, c_1, c_2, c_3$. Which number theory result can you use to obtain m^3 ? Deduce an algorithm to recover original message m.

Implement your algorithm. Hint: you can use the following import in `python`:

```
from sympy.ntheory.modular import crt
```

Note that you will need an implementation of cube root computation. Here it is:

```
def find_cube_root(n):
    lo = 0
    hi = n

    while lo < hi:
        mid = (lo + hi) // 2
        if mid**3 < n:
            lo = mid + 1
        else:
            hi = mid

    return lo
```

Let $e, n_1, n_2, n_3, c_1, c_2, c_3$ defined with :

$$\begin{aligned}
e = \quad & 3 \\
n_1 = \quad & 1154122453763259786013958396103535116763552806276696363118047423057773402 2\\
& 7893986431266777562253559076969010775854919542926778130322314104629015431 1\\
& 8762741872527713560619970911812956602184738183799683732007973420770977941 6\\
& 3663725052477982103879990759744692130726737385651844733853373404803583899 2\\
& 1784528629123 \\
n_2 = \quad & 8381666572009931103929884349219520528663806631040738137583915771088984769 7\\
& 2859654210085616089686477874136156625991190594804058521926789396653008155 3\\
& 4361811288419980986186399388506898972673503295417760263791980513079430701 7\\
& 9487172634485060765669419064873605700047591659122561176379236538860974873 2\\
& 414855319033 \\
n_3 = \quad & 1266979498633656547983661955021482836061136310399697502063177801979267838 9\\
& 1497255150089185907663345011031773702287375290195151108225428925532154658 3\\
& 5693313062602338951811278021374222265368384027983701332381937427636425787 2\\
& 7325392578673888298199563545027171658895306787607259114412416237420317858 6\\
& 8804404984937
\end{aligned}$$

$$
\begin{aligned}
c_1 = \ & 8903151370207932635942941592834283173826053287107373287045343129934711090 5\\
& 8098973053857746702655841300989938130820191499285001360127379026034950545 3\\
& 4383904327690416995531969548503690092288074479279161472100152130087011096 3\\
& 0789586564333708168782994405494771090292565333368738228488563958445098737 9\\
& 176160905778\\
c_2 = \ & 4856350068324757807467211369441194034751053506402294084924059268203150126 1\\
& 2340713826518502335470567731586271950963495567126868577641521819364790862 2\\
& 8034351385660986004644658506196361503171589514828358336943284964614852347 6\\
& 9045508115754467736767773382223299853311406570990753847697057658306079868 2\\
& 628023354138\\
c_3 = \ & 7183755944681305757660989645377722554953541398134501290194625009959307034 6\\
& 2443023781404205546149405223365954216304147045112050783830183193109093547 1\\
& 9304595934359038392134464671585721523775648566448722908296350109791579188 9\\
& 6582890705881492551433839396425197514907790927273798242269446710248351870 3\\
& 74137882163
\end{aligned}
$$

What is the size of n_i and c_i in bits ? Use your previous algorithm and your RSA implementation to find the original plaintext !