Prof. Dr. Heiko Knospe **Summer Semester 2025**

Cryptography Lab 1

Preparation: Study the slides on *Fundamentals*, work on the associated exercises and get familiar with the basics of Python programming.

The Lab uses the mathematics software SageMath and Jupyter notebooks.
(`http://www.sagemath.org`).

Make sure to evaluate your input (Shift-Enter) and to start a new cell **after each command** or after each definition of a function. **Do not write** multiple commands in a single cell unless it is really necessary! Fully document your code (almost every line) by using the `#` character in code cells or by formatting a cell as Markdown.

Throughout this lab, replace X by your 8-digit student matriculation number .

1. Start SageMath, open a new worksheet and rename it to X. Write your full name and X into the first cell:

```
# Name: ...
# Matriculation Number: X
```

Perform some elementary modular operations using SageMath. Do not forget to use new cells (Shift-Enter) and to save your worksheet regularly. Set $n = X{*}X{*}X$ (the cube of X), $a = X19898X65432098772$ and $b = 55432098771234567891433336543209877$.

(a) Find the prime factor decompositions of n, a and b. Use the `factor` command.

(b) Compute $a{-}b \bmod n$, $ab \bmod n$, $a^b \bmod n$, using `mod(..,..)` and `power_mod(..,..,..)`. Try to compute `a**b` and `mod(a**b,n)`. Why does this problem not occur with `power_mod(..,..,..)` ?

(c) Are a or b invertible modulo n ? Why or why not? Compute $(a \bmod n)^{-1}$ and $(b \bmod n)^{-1}$, if possible, and use `inverse_mod(...,...)`.

2. Implement the Extended Euclidean Algorithm (see lecture slides on Fundamentals). Complete the definition of the SageMath function `EuclAlg` listed below. Note that correct indentation is extremely important in Python. Replace the variable `gcd` by `egcd` since `gcd` is a built-in function.

```
def EuclAlg(a,b):
    x0=1;x1=0;y0=0;y1=1;sign=1;
    while b!=0:
        r=a%b
        q=a//b
        ...
    ...
```

Extend the code so that the number m of runs of the `while` loop is counted, and `return(egcd,x,y,m)`.
Compute `EuclAlg(a,n)` and `EuclAlg(b,n)` using the numbers from Task 1.
Then compute $(b \bmod n)^{-1}$ with the `EuclAlg` function.
Hint: Take b, n as input of `EuclAlg` and use `mod(x,n)`.

3. Write a function `units` which prints out the invertible integers mod n and their corresponding inverse, i.e., the elements of the group $\mathbb{Z}_n^*$ and the multiplicative inverse of each group element. The function shall return $ord(\mathbb{Z}_n^*) = \varphi(n)$. Do not use the built-in functions, but instead use `EuclAlg` to compute the greatest common divisor and the inverse.

```
def units(n):
     for i in range(1,n):
          if EuclAlg(i,n)[0]==1:
            ...
```

Check the function with $n = DD \cdot MM$, the product of the day and month of your birthday.

4. Set up an RSA cryptosystem and encrypt a message. First, generate two 1536-bit random primes. This may take some time.

```
p=random_prime(2**1536)
q=random_prime(2**1536)
```

Set $n = pq$ and $phi = \varphi(n) = (p-1) \cdot (q-1)$. Print out p, q, n and $\varphi(n)$.
Choose $e \geq X$ as small as possible such that $\gcd(e, \varphi(n)) = 1$. Set (e, n) as your public RSA key. Compute your private RSA key:

$$d \equiv (e \bmod \varphi(n))^{-1}$$

Verify that $ed \equiv 1 \bmod \varphi(n)$. Print out d. Now suppose that the integer $m = XXXXX$ is the given plaintext, i.e., five copies of your matriculation number. Compute the RSA ciphertext $c = m^e \bmod n$. Decrypt the ciphertext by computing $c^d \bmod n$ and compare the result with the plaintext m.

Hint: SageMath distinguishes between residue classes and integers. Use `ZZ(d)` to transform the residue class d into an integer.

Remark: The above schoolbook RSA algorithm has weaknesses. Randomized versions of RSA, e.g., RSA-OAEP, should be used instead. This will be discussed later in the course.

5. Conduct a Diffie-Hellman key exchange between Alice and Bob. They use the Diffie-Hellman group `ffdhe3072` from `https://tools.ietf.org/html/rfc7919`. Set

$$p = 2^{3072} - 2^{3008} + (\lfloor 2^{2942} \cdot e \rfloor + 2625351) \cdot 2^{64} - 1$$

where $e = \exp(1)$. Use the SageMath `floor( )` function. Print out `hex(p)` and compare the result with RFC 7919. Then set $g = 2$. Alice and Bob select private random keys a and b between 1 and $p-1$. A random integer less than p can be generated with `ZZ.random_element(p)`. Compute their public keys $A = g^a \bmod p$ and $B = g^b \bmod p$. Print out A and B. Then Bob computes $A^b \bmod p$, and Alice computes $B^a \bmod p$. Compare their results. How do you explain that?

6. Conduct a Diffie-Hellman key exchange similar as in Task 5 above, but now using the *Elliptic Curve 25519* (see RFC 7748 `https://tools.ietf.org/html/rfc7748`). Set

$$p = 2^{255} - 19$$
$$u = 9$$
$$v = 14781619447589544791020593568409986887264606134616475288964881837755586237401$$

Define the elliptic curve $E : y^2 = x^3 + 486662x^2 + x$ over $GF(p)$:

$$E = EllipticCurve(GF(p), [0, 486662, 0, 1, 0])$$

Define the point $g = (u, v)$ on the curve. One obtains an error message if the point does not lie on the curve.

$$g = E(u, v)$$

The points on E form a group and g generates a cyclic subgroup (the details will be explained later and are not important for this lab). Determine the order of g and print out the hexadecimal value `hex(n)`.

$$n = g.order()$$

For an Elliptic-Curve Diffie-Hellman key exchange, generate two random private numbers a and b between 1 and $n-1$ (similar as in Task 5). Compute the public keys:

$$A = a * g$$
$$B = b * g$$

Print out the points A and B. After exchanging A and B, both partners can compute the shared secret key. Compute $a * B$ and $b * A$ and compare the result. The first coordinate (x-coordinate of the shared secret key) is usually hashed and then used as a shared secret.

Compare the running times to compute the public keys and the shared secret key in Tasks 5 and 6. Use the `%time` command. Furthermore, compare the key lengths in Tasks 5 and 6.

Completion: Fully document your code. Prepare the final version. Make sure that all cells are evaluated and the results are human-readable and documented; then save the notebook. Upload the file `X.ipynb` to Moodle.

Cryptography Lab 2

In this lab, you write Python code to encrypt and to decrypt data with AES in GCM mode. Furthermore, the randomness of the counter mode ciphertext is investigated.

This lab uses Python and Jupyter notebooks. You may use the **SageMath** installation from Lab 1. Click on **New** and open a **Python 3** notebook. Then install the package `pycryptodome`:

```
!pip install pycryptodome
```

Alternatively, you may use any Python 3 / Jupyter installation. Make sure that you install `pycryptodome`.

Hint: After installing the `pycryptodome` package, it is usually necessary to restart the kernel of the notebook (via the Jupyter menu) or to restart Jupyter.

Let X be your matriculation number. Rename the notebook to X`-lab2.ipynb` and save your work regularly. Write your full name and X into the first cell:

```
# Name: ...
# Matriculation Number: X
```

Make sure to evaluate your input (Shift-Enter) and to use a new cell **after each command** or after each definition of a function. **Do not write** multiple commands into a single cell unless it is really necessary! Fully document your code .

Using the code or adapting the code of other students is **not acceptable**.

1. a) Write a function that encrypts and authenticates `data` with a `key` and returns the ciphertext, the nonce and the tag. Plaintexts, ciphertexts and keys are Python `bytes` objects which can be converted into an integer `list`.

Import several modules:

```
import secrets
import random
import binascii
import matplotlib.pyplot as plt
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
```

Define a function `AES_encr_GCM` which takes a `key` and `data` as input. The function first generates a random nonce:
`mynonce = secrets.token_bytes(12)`
Now construct a new AES ciphering object for GCM mode encryption:
`cipher = AES.new(key, AES.MODE_GCM, nonce=mynonce)`
Then `data` is encrypted and authenticated:
`ct, tag = cipher.encrypt_and_digest(data)`
Your function `AES_encr_GCM` finally returns the `nonce`, the ciphertext `ct` and the `tag`.

b) Implement the corresponding decryption and verification function `AES_decr_GCM`. What is input to that function? Construct a AES `cipher` object for GCM mode decryption. Then call the function `cipher.decrypt_and_verify`, which takes the ciphertext and the tag as input and gives the plaintext if the verification was successful. Finally, your function `AES_decr_GCM` returns the plaintext.

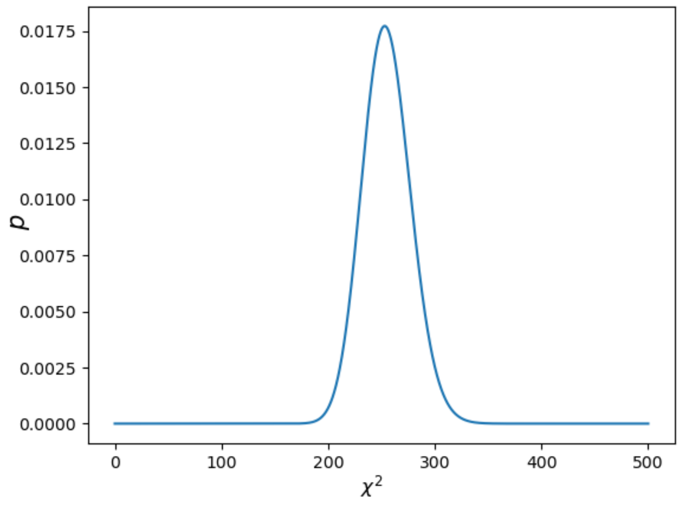
c) Generate a random AES key using `secrets.token_bytes(n)`, where n is the number of bytes. Which byte-lengths are supported by AES? Define a list of integer values containing your matriculation number, e.g. `matr=[1,2,3,4,5,6,7,8]`. Now extend the list such that it contains 1000 copies of your matriculation number. Plot the list using `plt.plot(mydata,'.')`. Then convert the list into a byte array with `bytes(mydata)` and encrypt the data using your function `AES_encr_GCM`. Print out the key, the nonce and the tag as a list of integers using `list( )`. Give the lengths of plaintext, key, ciphertext, nonce and tag. Then decrypt the ciphertext and compare the result with the plaintext.

d) Convert the ciphertext into a list called `mylist`. Plot the ciphertext `mylist`. Now modify the first entry `mylist[0]`. Convert the list back into a byte array using `ctmod=bytes(mylist)` and try to decrypt the modified ciphertext. How do you explain the result? Now improve your decryption function `AES_decr_GCM`: use `try` and `except` in the decryption function in order to catch a possible decryption error. Test the function, revert the change in the ciphertext and test the function again.

2. Generate a one megabyte all-zero plaintext (use `data=bytes(1000000)`) and encrypt the plaintext in GCM mode using a random key. Then analyze the uniform distribution of the AES ciphertext bytes by computing the χ^2 test statistic with 255 degrees of freedom. Let N be the ciphertext length in bytes (without nonce and tag) and define $E = \frac{N}{256}$. Determine the occurrence $N[i]$ of each byte value $i = 0, 1, \ldots, 255$. Print out the values $N[0], N[1], \ldots, N[255]$. Then compute the χ^2 test statistic:

$$\chi^2 = \sum_{i=0}^{255} \frac{(N[i] - E)^2}{E}$$

Compare your result with the χ^2_{255} probability density function (with 255 degrees of freedom) that is shown below. χ^2 is a goodness-of-fit test for the uniform distribution. Repeat the GCM mode encryption five times and compute the χ^2 statistic. How do you explain your results?



Probability density function χ^2_{255} with 255 degrees of freedom.

Completion: Fully document your code. Make sure that all cells are evaluated; then save the notebook. Upload the file `X-lab2.ipynb` to Moodle.

Cryptography Lab 3

In this lab, you write Python code for Elliptic Curve Diffie Hellman (ECDH) key encapsulation and hybrid ECDH/AES encryption.

This lab uses Python 3 and Jupyter notebooks, as in the preceding lab. Make sure that `pycryptodome` is installed.

Let X be your matriculation number. Rename the notebook to `X-lab3.ipynb` and save your work regularly. Write your full name and X into the first cell:

```
# Name: ...
# Matriculation Number: X
```

Evaluate your input (Shift-Enter) and start a new cell **after each command** or after each definition of a function. **Do not write** multiple commands into a single cell unless it is really necessary! Do not print out large binary data. Fully document your code.

Preparation: Study the chapters on symmetric encryption, message authentication and key establishment.

1. Import several modules:

```
import secrets
import random
from Crypto.Cipher import AES
from Crypto.PublicKey import ECC
from Crypto.Hash import SHA256
from Crypto.Protocol.KDF import HKDF
from Crypto.Protocol.DH import key_agreement
from Crypto.Random import get_random_bytes
```

a) Get your functions `AES_encr_GCM` and `AES_decr_GCM` for authenticated encryption and decryption from the last lab.

b) Generate a random secret key with the maximal key length supported by AES and a random plaintext of $1,000,000$ bytes. Use the functions `secrets.token_bytes(N)` for the key and `get_random_bytes(N)` for the plaintext. Run your encryption and decryption encryption functions and check that decryption is correct (do not print out).

2. Implement ECDH key encapsulation and decapsulation.

a) Write a HMAC-based *Extract-and-Expand Key Derivation Function* (HKDF, see RFC 5689), which generates a pseudorandom 32-byte key from an input key x.

```
def myhkdf(x):
        salt=...
        return HKDF(x, 32, salt, SHA256, 1)
```

Set `salt` to a `bytes` object that contains your matriculation number. Test the function with a binary input key x of length 12 bytes. Which is the output length?

b) Suppose your communication partner Bob has a static (long-term) elliptic curve key pair.
`B = ECC.generate(curve='Ed25519'); Bpub=B.public_key()`
Print out the public key.

c) For key encapsulation, we generate an ephemeral key pair.
`A = ECC.generate(curve='Ed25519'); Apub=A.public_key()`

d) We send our public key `Apub` (the encapsulated session key) to Bob and derive the session key:
`sessionkey = key_agreement(eph_priv=A, static_pub=Bpub, kdf=myhkdf)`
Print out the session key. Use the method `hex()`. How is the session key computed and which is the role of the key derivation function?

e) How does Bob derive the session key (decapsulation)? Adapt the input of the above `key_agreement` function and compute `sessionkey2`. Note that our keys `Apub` and `A` are ephemeral and Bob's keys `Bpub` and `B` are static. Check that the derived session keys are identical.

3. a) Write a function `HybridEnc(publickey, data)` for hybrid ECDH and AES-GCM authenticated encryption for data of any length. The hybrid encryption function takes the static public key of the recipient and the plaintext data as input. Generate an ephemeral key pair. Derive the session key and encrypt the plaintext with that key. Output the encapsulated session key, i.e. the ephemeral public key, the nonce, the ciphertext and the tag.

b) Write a function `HybridDec` for hybrid ECDH and AES-GCM decryption. What is the input to that function and why? Decapsulate the session key and decrypt the ciphertext. Output the plaintext,

c) Test the correctness of hybrid encryption and decryption using Bob's static key pair (Task 2) and the plaintext from Task 1.

d) Now modify one byte of the ciphertext (see last lab). What happens if you decrypt the modified ciphertext?

Completion: Fully document your code and answer the questions. Write the answer as comment or as Markdown cell into your notebook. Make sure that everything is readable and all cells are evaluated; then save the notebook. Upload the file `X-lab3.ipynb` to Moodle.