Louis-Marie LOE - 20251348
Master-Minds 2025
INSA-Lyon 30.11.2025

Cryptography & Blockchain – 5IF, INSA Lyon – LAB DES

Structure of the current Report:

1. **Feistel Cipher → feistel_cypher_final.py**
2. **Test Feistel Cipher → feistel_cypher_final.py**
3. **Full DES Implementation → des_implementation_final.py**
4. **DES performance testing → des_performance_test_final.py**
5. **DES performance comparison: My DES vs Library DES**
   → des_performance_comparison_final.py
6. **Frequency analysis attack Caesar vs DES**
   → des_vs_feistel_performance_test_final.py

→ Attachments: 4 python files (feistel_cypher_final.py, des_implementation_final.py, des_performance_comparison_final.py, des_vs_feistel_performance_test_final.py)

1. Feistel Cipher

DES: Design your own symmetric-key encryption and decryption algorithms based on the Feistel cipher and implement them. You may use any programming language e.g. Python, C++, Java. Note: You may select any trivial round function F and subkey generation (key schedule) algorithm for this exercise. For example, you could use a simple XOR operation as the round function and you could use the same subkey for each of the rounds.

1.1 Component	*Value (Hex Bytes)*
Plaintext	*41 42 43 44 ("ABCD")*
L0	*41 42*
R0	*43 44*
Key K	***01 01***

1.2 4-Round Feistel Encryption - with Python Function Mapping

Round	***Li (Input Left)***	***Ri (Input Right)***	***Computation (Ri = Li-1 XOR K)***	***XOR Result***	***Output (Li, Ri)***	***Python Function Used***
0	*41 42*	*43 44*	*—*	*—*	*L0 = 41 42R0 = 43 44*	*(split in feistel_encrypt)*
1	*41 42*	*43 44*	*(41 42) XOR (01 01)*	*40 43*	*L1 = 43 44R1 = 40 43*	*feistel_round(L, R, K) → uses xor_bytes*
2	*43 44*	*40 43*	*(43 44) XOR (01 01)*	*42 45*	*L2 = 40 43R2 = 42 45*	*feistel_round(L, R, K)*
3	*40 43*	*42 45*	*(40 43) XOR (01 01)*	*41 42*	*L3 = 42 45R3 = 41 42*	*feistel_round(L, R, K)*
4	*42 45*	*41 42*	*(42 45) XOR (01 01)*	*43 44*	*L4 = 41 42R4 = 43 44*	*feistel_round(L, R, K)*

1.3 Final Output *Step*	*Value*	*Python Function Used*
Final Swap (R4 L4)		

Round	Li (Input Left)	Ri (Input Right)	Computation (Ri = Li-1 XOR K)	XOR Result	Output (Li, Ri)	Python Function Used
Ciphertext		43 44 41 42	Returned by feistel_encrypt			

1.4 Algorithm Step	Python Function
XOR operation	xor_bytes(a, b)
One Feistel round	feistel_round(L, R, K)
Entire encryption loop	feistel_encrypt(plaintext, key)
Final concatenation	end of feistel_encrypt

1.5 Python Code → feiste_cipher_final.py

2. Feistel Testing

Generate a random secret key. Use your implementation of the Feistel cipher to encrypt several plaintext messages. Decrypt the ciphertext to verify the correctness of the cryptosystem.

2.1 Example 1

Message to encrypt: "ABCD" → bytes: 41 42 43 44
Key: 01 01
Rounds: 4
Feistel rule: L(i+1) = R(i), R(i+1) = L(i) XOR K

```
Round   Li      Ri      (Before XOR)        (After XOR)
------------------------------------------------------------------
0    41 42   | 43 44    (41 42 XOR 01 01)    40 43
1    43 44   | 40 43    (43 44 XOR 01 01)    42 45
2    40 43   | 42 45    (40 43 XOR 01 01)    41 42
3    42 45   | 41 42    (42 45 XOR 01 01)    43 44
4    41 42   | 43 44    (41 42 XOR 01 01)    40 43
------------------------------------------------------------------
```

Ciphertext = R4 | L4 = 43 44 | 41 42

2.1 Example 2

Message: b"ABCDEF12" *→ Bytes: 41 42 43 44 45 46 31 32*
Key used (4 bytes per half): 01 01 01 01
Rounds: 4
Feistel rule: L(i+1) = R(i), R(i+1) = L(i) XOR K

```
Round   Li          Ri          (Before XOR)            (After XOR)
------------------------------------------------------------------------
0    41 42 43 44 | 45 46 31 32   (41 42 43 44 XOR 01 01 01 01)   40 43 42 45
1    45 46 31 32 | 40 43 42 45   (45 46 31 32 XOR 01 01 01 01)   44 47 30 33
2    40 43 42 45 | 44 47 30 33   (40 43 42 45 XOR 01 01 01 01)   41 42 43 44
3    44 47 30 33 | 41 42 43 44   (44 47 30 33 XOR 01 01 01 01)   45 46 31 32
4    41 42 43 44 | 45 46 31 32   (41 42 43 44 XOR 01 01 01 01)   40 43 42 45
------------------------------------------------------------------------
```

Ciphertext = R4 | L4 = 40 43 42 45 | 41 42 43 44

2.3 Python Code → festel_cipher_final.py

3. DES Full Implementation

Implement the DES 56-bit symmetric-key encryption and decryption algorithms. Implement the round function F and subkey generation algorithm as used by DES. The round function involves an expansion operation, substitution (using S-boxes), and permutation (using P-boxes). The subkey generation algorithm involves circular left shift operations and a contraction permutation operation (called Permuted Choice 2). You may look at the following or any other guide for a description of DES:

3.1 DES Data Flow - Encryption Path – with Python Function Mapping

Step	*Operation / Stage*	*Input → Output*	*Python Function Used*
1	*Plaintext Input*	*64-bit plaintext block*	*(given as input)*
2	*Initial Permutation (IP)*	*64 bits → 64 bits*	*permute(..., IP)*
3	*Split into L0 and R0*	*64 bits → 32-bit L0, 32-bit R0*	*(done inside des_encrypt_block)*
4	*16 Feistel Rounds*	*(L, R, Ki) → new (L, R)*	*Loop inside des_encrypt_block*
4a	*Round Function F*	*R (32 bits) & Ki (48 bits) → 32 bits*	*f_function(R_bits, round_key)*
4b	*Expansion E*	*32 bits → 48 bits*	*permute(..., E)*
4c	*XOR with Round Key*	*48 bits → 48 bits*	*b ^ k inside f_function*
4d	*S-box Substitution*	*48 bits → 32 bits*	*Inside f_function (loop over S_BOX)*
4e	*P-Permutation*	*32 bits → 32 bits*	*permute(..., P)*
5	*Swap Halves (R16*	*L16)*	*32/32 bits swapped*
6	*Final Permutation (FP)*	*64 bits → 64 bits*	*permute(..., FP)*
7	*Ciphertext Output*	*64-bit ciphertext block*	*Returned by des_encrypt_block*

3.2 DES Key Schedule - Key Expansion Path – with Python Function Mapping

Step	*Operation / Stage*	*Input → Output*	*Python Function Used*
1	*Original Key Input*	*64-bit key*	*(given as input)*
2	*Permuted Choice 1 (PC-1)*	*64 bits → 56 bits*	*permute(key_bits, PC_1)*
3	*Split into C0 and D0*	*56 bits → 28-bit C0 + 28-bit D0*	*Inside generate_round_keys*
4	*Left Circular Shifts*	*Ci-1/Di-1 → Ci/Di*	*left_rotate(bits, n)*
5	*Permuted Choice 2 (PC-2)*	*56 bits → 48-bit round key Ki*	*permute(CD, PC_2)*
6	*Round Key Output*	*48-bit Ki*	*Built inside generate_round_keys*
7	*Use Key in F-function*	*Ki → used in Feistel round*	*Passed into f_function(R, Ki)*

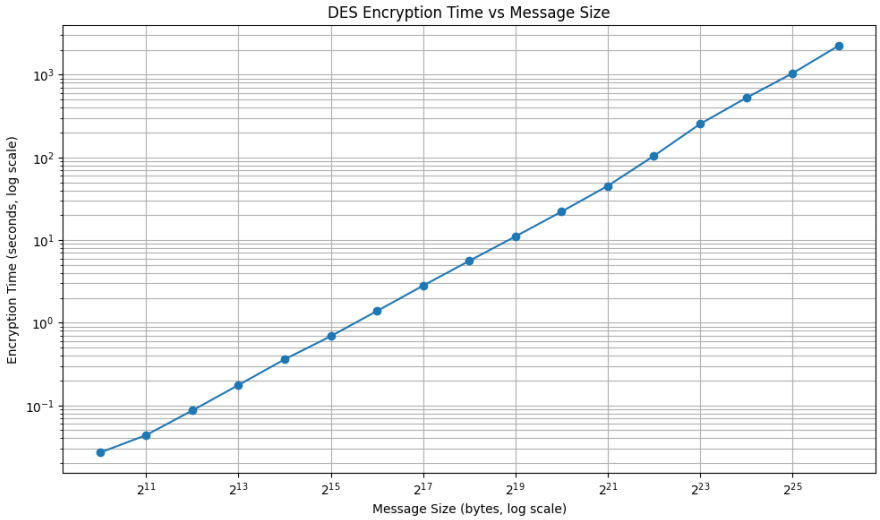
3.3 Python Code → des_implementation_final.py

4. Implementation Performance Analysis

Analyze the performance of your DES implementation for varying message sizes. Observe performance in terms of the processing time required for encryption. You may use the following message sizes: 2^10 B, 2^11 B, 2^12 B, …, 2^26 B

4.1. Plot of DES Encryption Time vs Message Size: Implemented DES executes in **O(n)** time

Observation: Message size is linearly proportional to encryption-time. if the message size doubles, the amount of time requires to encrypt it doubles. Encryption time scales linearly with message size.



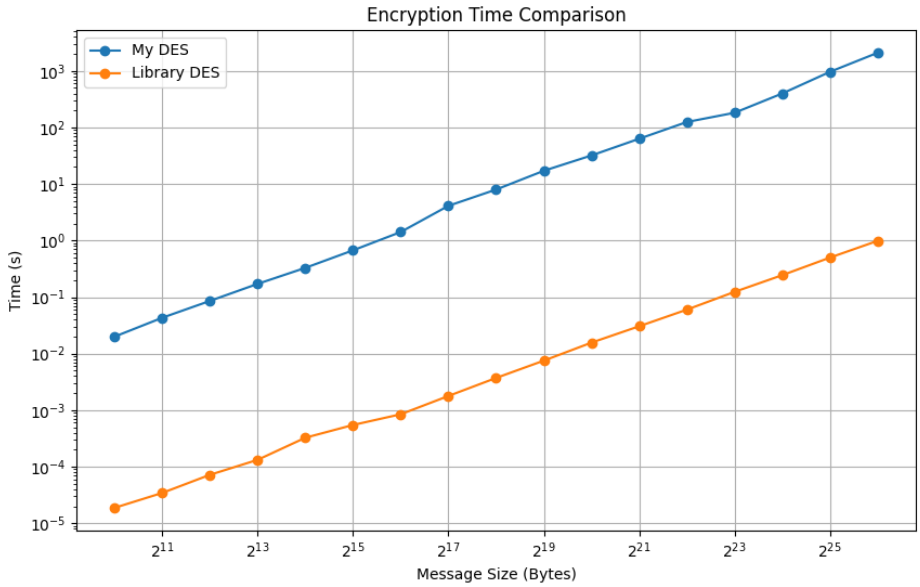
4.2 Python Code → des_performance_test_final.py

Additional exercises:

5. DES Performance Comparison: My DES vs Library DES

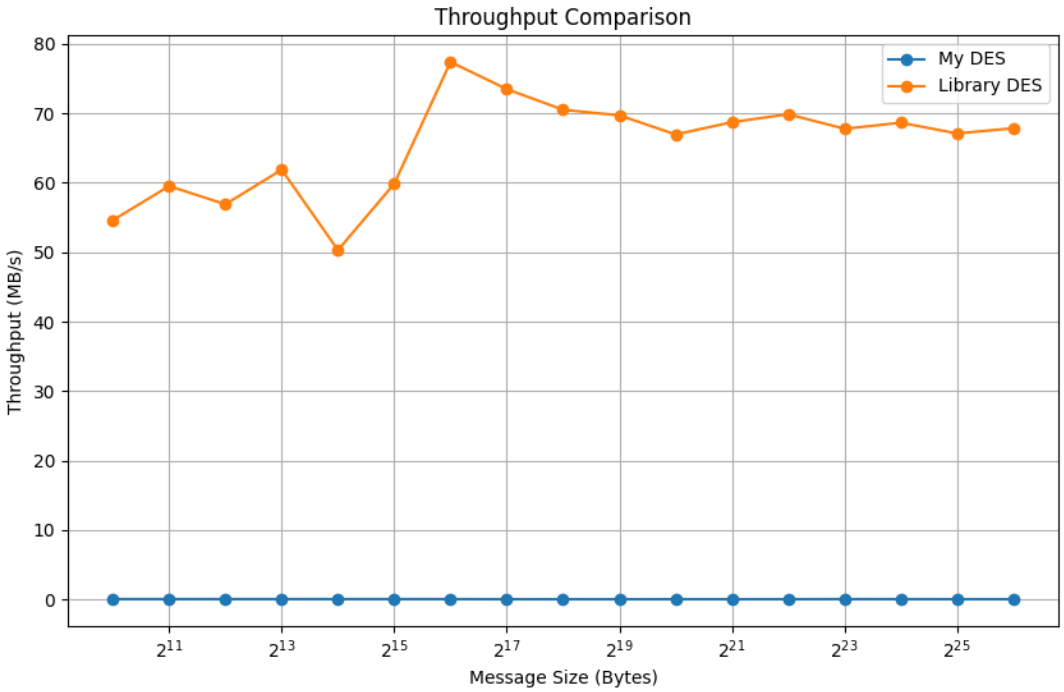
5. Use an existing library implementation of DES to encrypt plaintext messages. For example, in Python, you will find a DES implementation in the Crypto.Cipher package. Compare its performance with the performance of your own implementation.

5.1 Encryption Time Comparison



Observation: ***Both DES are linear, but the Library DES is about 1000 x faster than my DES: e.g.: for 2^23 Bytes (***8.388608 MB***), my DES takes over 100 seconds for encryption whereas the Library DES takes just 1/10 seconds for encryption***

5.2 Throughput Comparison



Observations: Using the same Laptop and Network, my DES maxes out and averages at 3.3MB/s while the DES from the library DES maxes out at close to 80 MB/s with an average around 63 MB/s.

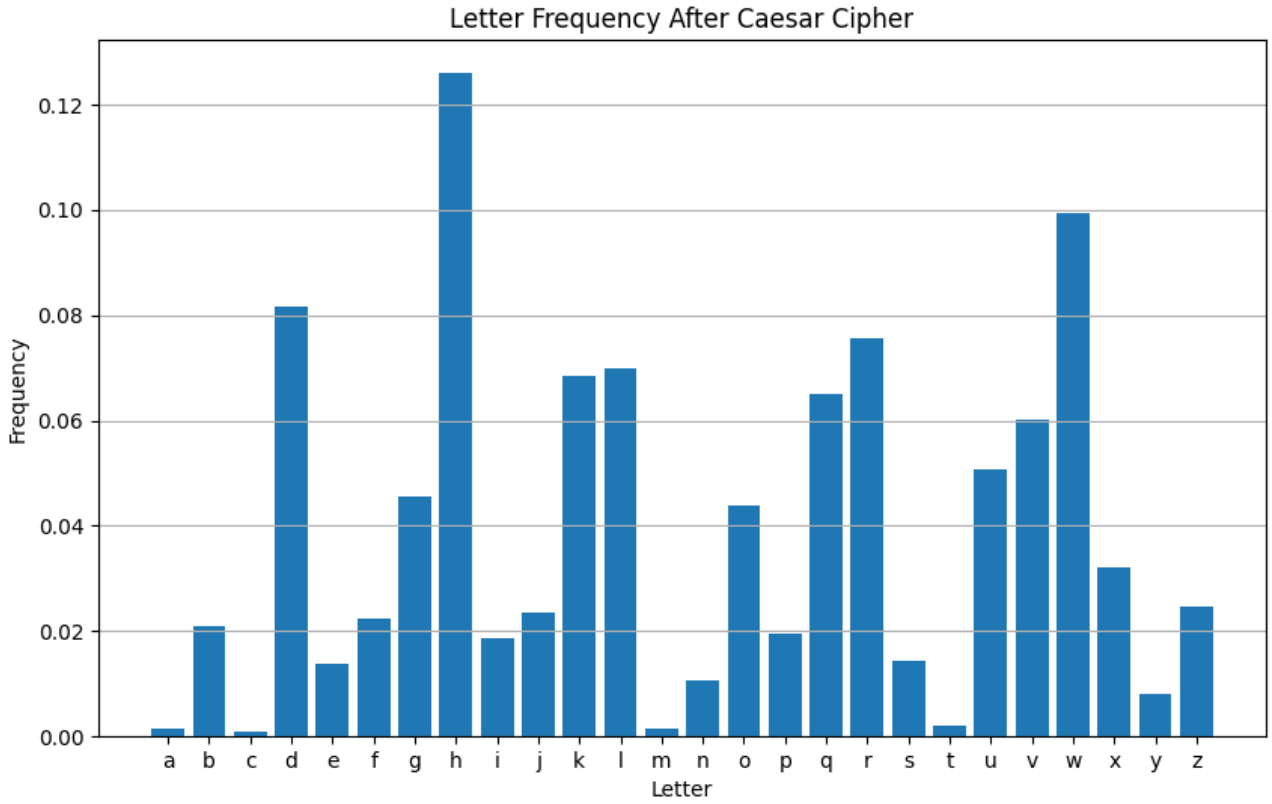
Conclusion: Both the time and throughput performance show the DES from the library is far more optimized as my DES implementation.

5.3 Python Code → des_performance_comparison_final.py

6. Feistel Cypher vs DES Cypher Frequency Analysis

Use a fairly large English text as the plaintext. Encrypt the entire text to ciphertext using a simple substitution cipher e.g. Caesar or Playfair. Run an English language alphabet frequency analysis attack on the ciphertext. Analyze the results. Then, encrypt the same plaintext with your DES implementation. Rerun the English language alphabet frequency analysis attack on the ciphertext. Analyze the results

6.1 Analysis of the Caesar Frequency analysis: ***The frequency graph shows that certain letters have higher frequencies than others, closely mirroring the natural distribution of English text. Thus, the Caesar cipher by preserving these plaintext frequencies is vulnerable to frequency analysis attacks. An attacker can exploit the uneven letter distribution to perform effective frequency-analysis attacks and recover the key.***



6.2 ***Analysis of the DES Frequency analysis:*** ***The frequency plots show that after DES encryption, all alphabet characters appear with nearly uniform frequency. This uniformity means that DES ciphertext does not preserve the statistical structure of the plaintext.***

As a result: Letter frequency patterns are destroyed, No meaningful linguistic information remains. An attacker cannot easily perform classical frequency-analysis attacks on DES. DES frequency analysis show: high diffusion and output randomness.