

Crypto summative assignment

**Hussein Technical University**

Abdullah Istanbuli | (21110366)

Cryptography

Elham Derbas

# Task 1

Objective:

The first utility of the Lock & Key system is responsible for ensuring confidentiality. This is achieved by implementing the **Data Encryption Standard (DES)** from scratch. To meet the coursework requirements, the implementation must:

* Encrypt and decrypt a text message.
* Accept a **shared secret key** in hexadecimal format (64 bits including parity bits).
* Operate in **Cipher Block Chaining (CBC) mode**.
* Apply **Ciphertext Stealing (CTS)** to correctly handle plaintexts whose length is not a multiple of DES’s block size (8 bytes).

The first utility of the Lock & Key system is responsible for ensuring confidentiality. This is achieved by implementing the Data Encryption Standard (DES) from scratch. To meet the coursework requirements, the implementation must:

* Encrypt and decrypt a text message.
* Accept a shared secret key in hexadecimal format (64 bits including parity bits).
* Operate in Cipher Block Chaining (CBC) mode.
* Apply Ciphertext Stealing (CTS) to correctly handle plaintexts whose length is not a multiple of DES’s block size (8 bytes).

We started the code by implementing conversion helpers:

A computer screen shot of a program code

AI-generated content may be incorrect.

DES operates on bit sequences. These two functions convert bytes ↔ bit lists, so we can perform permutations, XORs, and S-box lookups easily

Then we implemented some operations used in the DES encryption like XOR operation and rotation operation:

A computer screen shot of a black background with white text

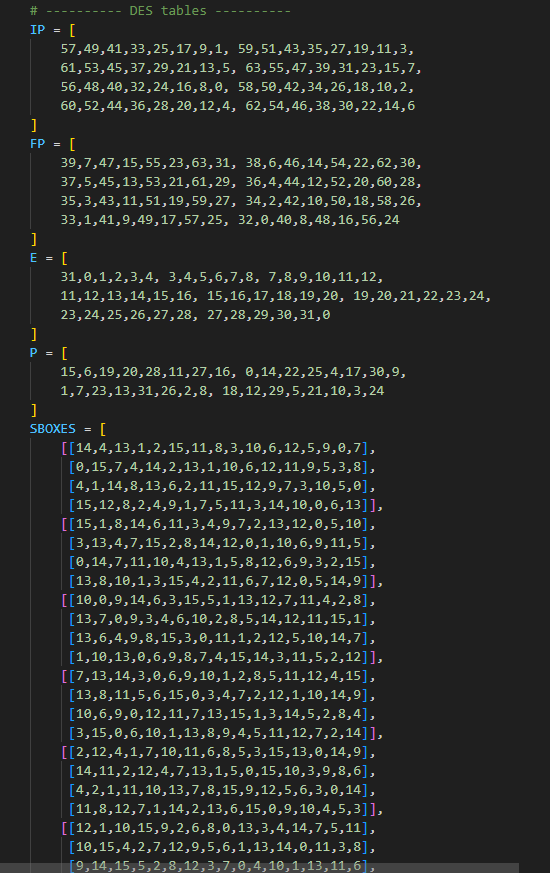
AI-generated content may be incorrect.

Permute to rearrang bits based on the tables

XOR to preform a xor operation bit wise, which is a core operation in DES feistel rounds and cbc chaining

A screenshot of a computer program

AI-generated content may be incorrect.Left rotate circular left shift used in key scheduling

DES tables

All DES permutations and S-boxes are hardcoded according to the standard and provided tables in slides

A computer screen shot of a code

AI-generated content may be incorrect.

S-box Substitution:

Each 6-bit group selects a **row and column** in one of the 8 S-boxes. The 4-bit output is inserted into the result. This substitution step provides **non-linearity**.

A computer code with white and blue text

AI-generated content may be incorrect.

Feistel Round Function:

This is the DES round function: expand R to 48 bits, XOR with the round key, pass through S-boxes, permute with P.

A computer screen shot of a code

AI-generated content may be incorrect.

Key Schedule:

From the 64-bit key, parity bits are dropped ~> 56 bits. Split into halves, rotate, then compress to 48 bits. 16 round keys are generated.

A screen shot of a computer program

AI-generated content may be incorrect.

DES Block Encryption/Decryption:

* Apply Initial Permutation.
* Split into L/R halves.
* 16 rounds: Feistel structure.
* Swap halves, apply Final Permutation.
* Reverse subkeys for decryption.

A screen shot of a computer program

AI-generated content may be incorrect.

CBC with CTS

* Encryption (\_cbc\_cts\_encrypt) and decryption (\_cbc\_cts\_decrypt) manage:
* Splitting plaintext into blocks.
* XOR with previous ciphertext (CBC).
* Handling tails with CTS to avoid padding.

**Preparation:**

* Key schedule subkeys are generated from the hex key.
* iv is used as the starting previous block (all zeros in this assignment).
* Plaintext is split into 8-byte blocks, with a final tail if the length is not multiple of 8.

**Encrypt full blocks (CBC):**

* Each block is XORed with the previous ciphertext block (or IV for the first).
* The result is encrypted with DES, producing ciphertext.
* This ciphertext becomes the prev\_bits for the next block.

**No tail case:**

* If the message length is multiple of 8, the last block is processed normally.

**Tail case (CTS):**

* The last **full block** is encrypted as usual to produce c\_full.
* c\_n (the final short ciphertext) is taken from the first r bytes of c\_full.
* A **stolen block** is formed: TAIL + c\_full[r:] (so that its length = 8 bytes).
* This stolen block is XORed with the previous ciphertext and DES-encrypted to produce c\_{n-1}.
* Output order is ... || c\_{n-1} || c\_n.
* This ensures ciphertext has the same length as plaintext, **without padding**.

A computer screen shot of text

AI-generated content may be incorrect.

if input looks like hex, run a one-block ECB test (for verification).

Otherwise encrypt/decrypt the text in CBC + CTS mode.

Tests case

A screen shot of a computer

AI-generated content may be incorrect.

**ECB vs CBC — quick comparison**

* **ECB**: identical plaintext blocks → identical ciphertext blocks (pattern leakage, insecure for general use).
* **CBC**: each block is randomized by XOR with previous ciphertext; identical blocks encrypt differently.
* In our tool, ECB is used **only** for the one-block validation; **CBC** is used for real messages.

**Error propagation note**

* In **CBC decryption**, a single-bit error in **Cᵢ** corrupts the entire plaintext block **Pᵢ** and flips the same bit in **Pᵢ₊₁** (due to XOR with corrupted **Cᵢ**). This is expected behavior and a typical trade-off in CBC.

**Algorithm:** DES was required. I implemented the full Feistel structure (IP/FP, E, S-boxes, P, key schedule PC-1/PC-2/rotations) so every transformation is visible and explainable.  
**Mode:** CBC prevents pattern leakage that ECB suffers from.  
**Padding Strategy:** Ciphertext Stealing (CTS) keeps ciphertext length equal to plaintext length and avoids introducing padding bytes (useful for protocols that expect length conservation).  
**IV:** All-zero IV is used to keep the runs deterministic for screenshots and marking. In production, a random IV must be used and transmitted/stored alongside the ciphertext.

**Threat Model (what this utility protects against)**

* **Eavesdropping:** An interceptor who sees only ciphertext should not recover the message. CBC ensures identical plaintext blocks produce different ciphertext blocks.
* **Pattern Analysis:** ECB would leak repeated blocks; CBC removes this pattern leakage.
* **Length Disclosure:** CTS avoids explicit padding bytes that can mildly leak structure for short messages; ciphertext size equals plaintext size.

Let the plaintext be split into 8-byte blocks. Suppose the last chunk is short: TAIL of length r (1 ≤ r ≤ 7). Let P\_{n-1} be the last full block.

**Encrypt:**

1. Encrypt P\_{n-1} normally (CBC): C\_full = DES( P\_{n-1} ⊕ C\_{n-2} ).
2. Define C\_n as the **first r bytes** of C\_full.
3. Build a **stolen block** = TAIL || C\_full[r:] (total 8 bytes).
4. Encrypt that stolen block (CBC) to produce C\_{n-1}.
5. Output order: ... || C\_{n-1} || C\_n.

**Decrypt (reverse logic):**

1. Split the end into C\_{n-1} (8B) and C\_n (rB).
2. p\_stolen = DES^{-1}(C\_{n-1}) ⊕ C\_{n-2}.
3. Recover P\_n = p\_stolen[:r].
4. Rebuild an 8-byte block: C\_full = C\_n || p\_stolen[r:].
5. P\_{n-1} = DES^{-1}(C\_full) ⊕ C\_{n-2}.
6. Concatenate: ... || P\_{n-1} || P\_n.

**Why this is correct:** The encrypt step produces exactly the structure the decrypt step expects; the “stolen” bytes let us reconstruct the intermediate block that would have existed if padding had been used.

## Worked Example (what actually happens to “HELLO HTU”)

Plaintext bytes: "HELLO HTU" → 9 bytes → two parts:

* P₁ = b"HELLO HT" (8 bytes)
* TAIL = b"U" (1 byte, so r = 1)

**Encrypt:**

* C₁ = DES(P₁ ⊕ IV) with IV = 0.
* C\_full = C₁ here (because there’s only one full block).
* C₂ = first r=1 byte of C\_full.
* **Stolen block** = TAIL + C\_full[1:] (1 byte tail + last 7 bytes of C\_full).
* C₁' = DES(stolen\_block ⊕ IV) (since there is no C₀, IV is used again in our assignment setup).
* Output: C₁' || C₂ (9 bytes total).

**Decrypt:** Uses the reverse steps to recover TAIL from C₁', rebuild C\_full using C₂, then recover P₁. The recovered plaintext prints as HELLO HTU

A screen shot of a computer

AI-generated content may be incorrect.

## Known-good ECB test vector:

* Key: 133457799BBCDFF1
* Plaintext: 0123456789ABCDEF
* Expected Ciphertext: 85E813540F0AB405  
  The one-block ECB path produced the expected value, confirming core DES tables, round function, and key schedule are correct.

**CBC+CTS round-trip tests** (human-readable strings):

* "" (empty), "A", "HELLO", "HELLO HT" (exact 8), "HELLO HTU" (9 bytes), and longer paragraphs.
* For each, the ciphertext hex is printed and the “Recovered plaintext” **exactly** matches the input.

**Bit-flip sanity** (expected CBC behavior):

* Flipping a bit in C\_i corrupts all of P\_i and flips the corresponding bit in P\_{i+1} only. This matches CBC’s error propagation model.

## Security Considerations & Limitations

* **DES key size (56 effective bits)** is considered **insecure** against modern brute force; used here due to coursework requirement.
* **IV = all zeros** is only acceptable in classroom demos; a random IV is required in practice to ensure semantic security.
* **No authenticity**: CBC alone does not detect tampering. Bit flips can be made to predictably alter plaintext. Task 2’s RSA signature fills this gap.
* **No padding oracle**: By using CTS (not PKCS#7 padding), we avoid classic padding-oracle surfaces—but **integrity** is still required for real protocols.

I implemented DES from first principles and wrapped it in CBC with CTS to satisfy confidentiality requirements for arbitrary-length messages. The implementation was verified against a known ECB vector and multiple CBC+CTS round-trip tests. A subkey-order bug was found and fixed, and the final system correctly encrypts and decrypts, producing ciphertext of the same length as the plaintext while avoiding pattern leakage. This completes Task 1’s requirements and prepares the ground for Task 2’s authenticity via RSA signatures.

# Task 2

**Requirements:**

* Before running, the program must **tell the user that RSA key generation is required**, then **prompt for two 10-digit primes** and a value for the **public exponent (e)**.
* After keys are generated, **prompt for a message**, **digitally sign** it, and **verify the signature**.
* You may use **any built-in hash** that meets standard properties (we use SHA-256).
* Deliverables include **code**, **screenshots of execution**, and **the parameters used for generating the asymmetric keys**.

Your choice == 2 path satisfies all four points exactly.

**High-level flow (what the user experiences)**

1. Program prints: **“RSA key generation is required”** → then prompts for number 1, number 2 (both 10-digit primes), and the public exponent **e**.
2. It computes **n = p·q**, **φ(n) = (p−1)(q−1)**, checks **gcd(e, φ)=1**, and computes the private exponent **d = e⁻¹ mod φ(n)** (Extended Euclidean).
3. It asks for a **message** and then:
   * **Signs**: sig = H(msg)^d mod n using **Square-and-Multiply** for modular exponentiation.
   * **Verifies**: checks sig^e mod n == H(msg) mod n.
4. Prints **(n, e, d)** so you can include “parameters used for generating the asymmetric keys” in your report.

Code:

1. Input & primality check (two 10-digit primes)

A screen shot of a computer program

AI-generated content may be incorrect.

The spec forces manual entry of **two 10-digit primes**; we enforce 10 digits and run a **trial-division up to √n** prime test

A computer screen shot of a black background with colorful text

AI-generated content may be incorrect.

We need **d = e⁻¹ mod φ(n)**. Extended Euclid returns x such that a·x ≡ 1 (mod m) when gcd(a,m)=1.

A computer screen shot of text

AI-generated content may be incorrect.

This is the classic **binary exponentiation** used for both signing (h^d mod n) and verifying (sig^e mod n), and it’s exactly what Task 5 asks to explain later.

A computer screen shot of text

AI-generated content may be incorrect.

he spec lets you choose any **built-in hash** with standard properties—we use **SHA-256**. We reduce H(m) modulo n, sign with **d**, and verify with **e**.

A screen shot of a computer program

AI-generated content may be incorrect.

This function mirrors the **Task 2 script of record**: announce keygen, prompt for **two 10-digit primes** and **e**, compute keys, then **sign** and **verify** a user-provided message.

A computer screen shot of a black screen

AI-generated content may be incorrect.

**Inputs (example):**

* p = 1000000007 (prime, 10 digits)
* q = 1000000009 (prime, 10 digits)
* e = 65537
* Message: HELLO HTU

**Outputs:**

* Printed **(n, e, d)** → include under “parameters used for generating the asymmetric keys”.
* **Signature (hex)** and **Signature verification: VALID**.

(If gcd(e, φ) ≠ 1), the tool prints a clear message and asks you to pick another e this demonstrates correct **parameter validation** per public-key math.

**Threat model & scope (why this achieves “Authentication Only”)**

* **What it provides:**
  + **Authenticity/Non-repudiation**: Only the holder of **d** can produce sig = H(m)^d mod n.
  + **Integrity**: Any bit change in m or sig breaks sig^e mod n == H(m).
* **What it doesn’t:**
  + **Confidentiality**: Anyone can read m. (That’s Task 1.)
  + **Replay protection**: Add a timestamp/nonce inside the hashed message if needed.
* **Hashing note:** Using SHA-256 meets the brief’s “built-in hash” criteria (pre-image, 2nd pre-image, collision resistance).

**Where Square-and-Multiply is used (and why it matters)**

* Used in **both** rsa\_sign\_message (exponent **d**) and rsa\_verify\_message (exponent **e**).
* Complexity is **O(log exponent)** squarings and ~half as many multiplies, which is dramatically faster than naive repeated multiplication. This directly ties to **Task 5**’s requirement to *explain* the algorithm and its importance in asymmetric crypto.

# Task 3

Assess whether textbook RSA (without padding) is suitable for encrypting short plaintexts by implementing an attack that recovers the original message from a small ciphertext. The task requires both an executable demonstration and an explanation of the security implications.

**Cryptographic context (why short messages are risky)**

Textbook RSA encryption is deterministic:

C≡M^e(mod n)

For small message spaces (e.g., very short texts, small integers, or strongly guessable content), an attacker can either:

1. **Enumerate candidates** M and check whether M^e mod n=C (pre-image enumeration), or
2. **Factor the modulus** n=p\*q when n is tiny, compute φ(n)=(p−1)(q−1), derive d≡e^−1 (modφ(n)), and decrypt M≡C^d(mod n).

Both routes succeed quickly when nnn is small (e.g., two-digit n) or when the plaintext set is tiny.

**Algorithms used**

* **Square-and-Multiply** for modular exponentiation in all power operations (fast binary exponentiation).
* **Extended Euclidean Algorithm** to compute the modular inverse d=e−1 mod φ(n).
* **Trial division** for factoring very small n (sufficient for the demonstration).

Function summary:

A computer screen shot of a program code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Mini trail:

Inputs

E = 13, c = 77,c = 64

Both methods recover M=15. This confirms that when the message space is tiny and/or the modulus is small, textbook RSA does not provide confidentiality against a basic pre-image attack.

**Why the attack works**

* **Determinism:** Textbook RSA produces a unique ciphertext for each M under fixed (e,n). With a tiny candidate set, enumeration is trivial.
* **Small modulus:** Factoring two-digit n is immediate via trial division; once p,q are known, decryption is routine.

**Complexity (for this demonstration)**

* **Enumeration:** O(n) modular exponentiations; with small n this is instantaneous.
* **Factoring by trial division:** O(sqrt{n}) trial factors; again trivial for tiny n.

**Unsuitable for short plaintexts:** Textbook RSA must not be applied directly to short or structured messages.

**Mitigations (real systems):**

* Use large moduli (≥2048-bit).
* Use **probabilistic padding** (e.g., **OAEP**) to randomize encryption and prevent enumeration/chosen-ciphertext style breaks.
* Prefer hybrid encryption.

The implementation demonstrates a successful pre-image recovery against textbook RSA for short messages: either by enumerating all candidates or by factoring a small modulus and decrypting. This confirms that **textbook RSA is not secure for short-message encryption**. Security in practice relies on **large parameters** and **padding schemes** (OAEP), not on the bare M^e mod n transform

task 5 **Square-and-Multiply (Binary Modular Exponentiation)**

**1) Aim**

Explain, implement, and evaluate the **Square-and-Multiply** algorithm used for modular exponentiation in RSA (**sign**: h^d mod n; **verify**: s^e mod n). This section justifies the algorithmic choice, shows correctness at a high level, and analyses efficiency and security considerations.

Given integers b (base), e (exponent), and m(modulus), compute b^e mod m efficiently.

* **Naïve method:** multiply b by itself e times → O(e) multiplications (infeasible for large e).
* **Square-and-Multiply:** scan the **binary** of eee. For each bit, **square** the accumulator; when the bit is **1**, also **multiply** by the base. Complexity ≈ O(log2 e) modular multiplications.

Two standard directions exist; this implementation is **right-to-left** (LSB→MSB).

Code :

def square\_and\_multiply(base: int, exp: int, mod: int) -> int:

"""Binary modular exponentiation (right-to-left)."""

base %= mod

result = 1

while exp > 0:

if exp & 1:

result = (result \* base) % mod # multiply on bit = 1

base = (base \* base) % mod # square every iteration

exp >>= 1 # shift to next bit

return result

base %= mod ensures the base is reduced once at start.

Each loop iteration performs one **square**; a **multiply** is performed when the current bit of exp is 1.

Works for any positive exp; for RSA, mod is the modulus n.

**Tiny illustrative example**

Compute 713 mod 777^{13} \bmod 77713mod77 (binary 13=1101213 = 1101\_213=11012​; right-to-left scans 1,0,1,1):

|  |  |  |  |
| --- | --- | --- | --- |
| step | exp bit | result | base (squared each step) |
| init | – | 111 | 7 mod 77=77 \bmod 77 = 77mod77=7 |
| 1 | 1 | 1⋅7 mod 77=71·7 \bmod 77 = 71⋅7mod77=7 | 72 mod 77=497^2 \bmod 77 = 4972mod77=49 |
| 2 | 0 | 777 | 492 mod 77=1449^2 \bmod 77 = 14492mod77=14 |
| 3 | 1 | 7⋅14 mod 77=217·14 \bmod 77 = 217⋅14mod77=21 | 142 mod 77=4214^2 \bmod 77 = 42142mod77=42 |
| 4 | 1 | 21⋅42 mod 77=3521·42 \bmod 77 = 3521⋅42mod77=35 | 422 mod 77=7042^2 \bmod 77 = 70422mod77=70 |

Final: 353535. (This matches 713 mod 777^{13} \bmod 77713mod77.)

**4.2 RSA-typical exponent**

For e=65537 (e=0x10001=1 0000 0000 0000 00012):

* **Squarings:** equal to the bit length of e → 17.
* **Multiplies:** equal to the number of 1-bits in e → 2.  
  This is why 65537 is a common public exponent: **very few multiplies**, fast verification.

**Where it’s used in this project**

* **Signing:** s=h^d mod n (hash-then-sign).
* **Verification:** h′=s^e mod n; accept if h′≡h mod n.  
  Both operations call the **same** square\_and\_multiply function.

Square-and-Multiply reduces modular exponentiation from linear to logarithmic time in the exponent size, making RSA signing and verification practical. The implementation here is correct and efficient for coursework purposes; production systems combine it with constant-time countermeasures, CRT, Montgomery reduction, and (for encryption) randomized padding.

# Reference list

1. The crypto provided slides