International Institute of Information Technology Hyderabad

System and Network Security (CS8.403)

Lab Assignment 2:
Secure UAV Command and Control System with Asymmetric Keys and Digital Signatures
Hard Deadline: 10-02-2026, 11:59 PM
Total Marks: 100

Note:- *It is strongly recommended that no group is allowed to copy programs from others. Blind use of high-level cryptographic libraries (like OpenSSL wrappers or built-in ElGamal modules) is strictly forbidden. You must implement the modular arithmetic (Modular Exponentiation, Modular Inverse via Extended Euclidean) and ElGamal protocol logic manually.* ***Languages: C, C++, or Python only.***

1. Objective

The objective is to implement a secure, distributed UAV Command-and-Control (C2) system. The system consists of a **Mission Control Center (MCC)** and multiple **Drones**. Key components include manual ElGamal implementation, mutual authentication, session management, and group key aggregation for fleet-wide secure broadcasting.

2. Cryptographic Specifications (Manual Implementation)

Students must implement the following ElGamal primitives from scratch in `crypto_utils.py`:

- **Key Generation:** Select a large prime p ($SL \geq 2048$) and generator g. Private key $x \in [1, p-2]$, Public key $y = g^x \pmod{p}$.
- **Encryption (E_{KU}):** Given message m, select random $k \in [1, p-2]$. Ciphertext $C = (c_1, c_2)$ where $c_1 = g^k \pmod{p}$ and $c_2 = (m \cdot y^k) \pmod{p}$.
- **Decryption (D_{KR}):** $m = (c_2 \cdot (c_1^x)^{-1}) \pmod{p}$.

- **Digital Signature ($Sign_{KR}$):** Given $H(m)$, select random k such that $gcd(k, p-1) = 1$. Signature $\sigma = (r, s)$ where $r = g^k \pmod p$ and $s = (H(m) - xr)k^{-1} \pmod{p-1}$.
- **Signature Verification ($Verify_{KU}$):** Check $g^{H(m)} \equiv y^r r^s \pmod p$.

3. System Architecture & Concurrency

3.1 MCC Server Design

The MCC must handle multiple drones simultaneously using **Multi-threading** or **Asynchronous I/O**.

- **Main Thread:** Listens for new connections on a TCP port.
- **Drone Threads:** Upon a new connection, spawn a thread to handle Phases 1 and 2.
- **Fleet Registry:** A thread-safe data structure (e.g., a synchronized dictionary) that stores active `Drone_ID`, `Socket_Object`, and `Session_Key`.

3.2 Interface

The MCC must provide a live Command Line Interface (CLI) to:

1. `list`: Show all currently authenticated drones and their status.
2. `broadcast <cmd>`: Generate a Group Key, distribute it, and send an encrypted command to all drones.
3. `shutdown`: Close all sessions and exit.

4. Protocol Phases

Phase 0: Parameter Initialization (MCC $\rightarrow$ Drone)

The MCC acts as the "Root of Trust" and defines the current operational security parameters.

1. MCC selects $SL = 2048$ (or higher).
2. MCC generates/selects a prime p of length SL and a generator g.
3. MCC creates $M_0 = \langle p \parallel g \parallel SL \parallel TS_0 \parallel ID_{MCC} \rangle$.
4. **Why both SL and p?** The Drone first checks if $len(bin(p)) \approx SL$. Then, it checks if $SL \geq$ its internal hardcoded safety limit. If the MCC tries to use a weak 512-bit prime but claims $SL = 2048$, the drone must detect this inconsistency and abort.

Here,

- **Security Level (SL):** An integer representing the required strength (e.g., 2048 or 3072). It defines the bit-length of the prime p.
- **The Prime (p):** A large prime number such that $2^{SL-1} < p < 2^{SL}$.
- **The Generator (g):** A primitive root modulo p.

Phase 1: Mutual Authentication

Phase 1A: Drone Request (Drone → MCC)

1. Drone generates random 256-bit secret $K_{D_i,MCC}$ and nonce RN_i.
2. Drone encrypts $C_i = E_{KU_{MCC}}(K_{D_i,MCC})$.
3. Drone sends $\langle TS_i, RN_i, ID_{D_i}, C_i, Sign_{KR_{D_i}}(TS_i \parallel RN_i \parallel ID_{D_i} \parallel C_i)\rangle$.

Phase 1B: MCC Response (MCC → Drone)

1. MCC verifies signature, decrypts C_i to obtain $K_{D_i,MCC}$.
2. MCC generates RN_{MCC} and TS_{MCC}.
3. MCC encrypts the *same* key back to the drone: $C_{MCC} = E_{KU_{D_i}}(K_{D_i,MCC})$.
4. MCC sends $\langle TS_{MCC}, RN_{MCC}, ID_{MCC}, C_{MCC}, Sign_{KR_{MCC}}(TS_{MCC} \parallel RN_{MCC} \parallel ID_{MCC} \parallel C_{MCC})\rangle$.

Phase 2: Session Key Generation & Confirmation

1. Both parties derive $SK_{D_i,MCC} = \mathrm{H}(K_{D_i,MCC} \parallel TS_i \parallel TS_{MCC} \parallel RN_i \parallel RN_{MCC})$, where $H(\cdot)$ is the SHA-256.
2. **Drone Confirmation:** Drone sends $HMAC_{SK}(ID_{D_i} \parallel TS_{final})$.
3. **MCC Confirmation:** If HMAC matches, MCC registers the drone and responds with `OPCODE 50 (CONFIRM)`. If not, MCC sends `OPCODE 60 (MISMATCH)` and blocks the ID_{D_i}.

Phase 3: Group Key Establishment (Fleet Aggregation)

1. When MCC triggers a broadcast, it aggregates SK's of n active drones.
2. Calculate $GK = \mathrm{H}(SK_{D_1} \parallel SK_{D_2} \parallel \cdots \parallel SK_{D_n} \parallel KR_{MCC})$, where $H(\cdot)$ is the SHA-256.
3. MCC sends GK to each drone, encrypted via AES-256 in CBC mode using each drone's unique SK_{D_i}.
4. All subsequent messages use GK for AES-256 encryption and HMAC-SHA256 for integrity.

5. Protocol Opcodes

All messages must start with a 1-byte Opcode for protocol parsing.

Opcode	Type	Description
10	PARAM_INIT	Phase 0: Crypto Parameters and MCC Sig
20	AUTH_REQ	Phase 1A: Drone Authentication Packet
30	AUTH_RES	Phase 1B: MCC Proof of Decryption
40	SK_CONFIRM	Phase 2: Session Key Verification (HMAC)
50	SUCCESS	Handshake Complete
60	ERR_MISMATCH	Key or HMAC Verification Failed
70	GROUP_KEY	Phase 3: Distribution of GK
80	GROUP_CMD	Secure Broadcast (Encrypted via GK)
90	SHUTDOWN	Close all drone connections

7. Library Usage Policy

To ensure students understand the underlying mathematics of asymmetric cryptography, the following rules apply:

7.1 Permitted Libraries

You may use standard libraries for networking, concurrency, and basic cryptographic building blocks:

- **Networking/System:** `socket`, `threading`, `asyncio`, `select`, `struct`, `sys`, `time`.
- **Hashing & MAC:** `hashlib` (for SHA-256), `hmac` (for HMAC-SHA256).
- **Symmetric Encryption:** `pycryptodome` or `cryptography.hazmat` **only** for the raw AES-CBC block cipher (Phase 3).
- **Randomness:** `secrets` or `os.urandom` (for cryptographically secure random numbers).
- **Large Number Math (C/C++ only):** Students using C++ may use **GMP (GNU Multiple Precision Arithmetic Library)** to handle 2048-bit integer arithmetic. Python students must use Python's built-in arbitrary-precision integers.

7.2 Strictly Not Allowed Libraries

Using the following will result in a **zero mark** for the cryptographic portion of the lab:

- **High-Level Security Wrappers:** `ssl`, `paramiko`, `pyOpenSSL`, or Python's `secrets` (for anything other than generating random numbers).

- **Asymmetric Abstractions:** Any library that provides pre-built ElGamal, RSA, or ECC objects (e.g., `Crypto.PublicKey.ElGamal` or `cryptography.hazmat.primitives.asymmetric`).
- **Digital Signature Modules:** Any module that automates the signing/verification process (e.g., `Crypto.Signature.DSS`).
- **Key Exchange Frameworks:** Any library that implements Diffie-Hellman or automated key exchange logic.

The "Manual" Rule

You must manually write the functions for:

1. Modular Exponentiation ($a^b \pmod n$) and Modular Inverse.
2. The ElGamal Encryption/Decryption logic (c_1, c_2).
3. The ElGamal Signing/Verification logic (r, s).

8. Final Submission Files

- **crypto_utils.py:** Manual ElGamal, Modular Math, HMAC/AES wrappers.
- **mcc.py:** Concurrent server with interface.
- **drone.py:** Client-side protocol logic.
- **attacks.py:** A script to demonstrate:
  - **Replay Attack:** Re-sending Phase 1A.
  - **MitM Tampering:** Modifying the prime p in Phase 0 to see signature failure.
  - **Unauthorized Access:** An unknown Drone ID attempting to connect.
- **SECURITY.md:** Explain how the protocol ensures *Freshness* and *Forward Secrecy*.
- **README.md:** Performance logs (Modular Exponentiation time for 2048-bit primes).