# Introduction

This document outlines the design of the ECC Secure Communication Protocol (ESCP), intended for IOT devices mainly focusing on efficient and secure data exchange between clients and server in a networked environment. ESCP is designed to be lightweight, easy to implement and flexible enough to accommodate future extensions.

# Goals

Efficiency: Minimize bandwidth usage and processing overhead.  
Reliability: Ensure data integrity and order of messages.  
Security: Support for encryption and authentication.  
Extensibility: Allow for future enhancements without breaking compatibility.

# Protocol Overview

ESCP uses a message-based approach where each message is a discrete unit of communication. This use TCP/IP for transport layer link and as part of this communication we manage the session connection and termination reducing the number of messages type. Once connection is established using ESCP we can use APIs provided to send and receive messages.

3.1 Protocol States

The protocol is designed around several states that manage the connection lifecycle and the negotiation of security parameters:

* RESET: The initial state before any connection attempts is made.
* NO\_CONNECTION: Indicates that no current connection exists.
* HANDSHAKE: The state during which the handshake process is active, negotiating security parameters.
* HANDSHAKE\_FAILED: Indicates that the handshake process was unsuccessful.
* ACTIVE: The state indicating that a secure connection has been established and is active.
* CLOSE: Indicates that the connection has been closed.
  1. Handshake Stages

The handshake process is broken down into distinct stages, each representing a step in the negotiation between the client and server to establish a secure connection. The stages are as follows:

* CLIENT\_GREET: The initial step where the client sends a hello message to the server, indicating its supported cipher suites and compression methods.
* SERVER\_GREET: The server responds to the client's greeting with its own hello message, selecting the cipher suite and compression method to be used.
* SERVER\_KEY: The server sends its public key to the client for asymmetric encryption.
* SERVER\_GREET\_DONE: Indicates the server has finished its part of the negotiation, awaiting the client's response.
* CLIENT\_KEY: The client sends its public key to the server.
* SERVER\_CERTIFICATE: The server sends its digital certificate to the client for authentication.
* CLIENT\_CERTIFICATE: The client optionally sends its digital certificate to the server if mutual authentication is required.
* FINISHED: The final stage where both parties confirm the establishment of a secure channel.  
  1. Data Structures

We have different message types which will support the handshake communication process. For the first version we are combing different stages of the handshake communication process to have basic and simple coding structure. Which can be further split into smaller message type in the future if required. Data structure is as shown below

typedef struct HANDSHAKE\_MESSAGETYPE

{

    uint16\_t cipher\_suite\_length;

    cipherSuite cipher\_suite[10];

    uint8\_t compression\_methods\_length;

    unsigned char\* compression\_method;

}handshake\_hello\_message;

typedef struct HANDSHAKE\_KEY\_EXCHANGE

{

    uint16\_t key\_length;

    unsigned char public\_key[65]; // uncompressed byte

}handshake\_key;

typedef struct HANDSHAKE\_CERTIFICATE

{

    uint16\_t certificate\_length;

    unsigned char certificate[128]; // Length should be changed when actual implementation is done

}handshake\_certificate;

typedef union HANDSHAKE\_MESSAGE

{

    handshake\_hello\_message hello\_message;

    handshake\_key key;

    handshake\_certificate certificate;

}handshake\_message;

typedef struct HANDSHAKE\_MESSAGE\_DATA

{

    handshakeStages handshake\_message\_stage;

    uint16\_t protocol\_version;

    handshake\_message message\_data;

}handshake\_message\_data;

typedef struct PROTOCOL\_MESSAGE\_HANDSHAKE

{

    protocol\_state message\_type;

    uint16\_t session\_id;

    handshake\_message\_data protocol\_data;

}protocol\_message\_data;

* 1. Protocol Flow
* **Hello Message**: The handshake begins with the client sending a HELLO\_MESSAGE, listing preferred cipher suites and compression methods.
* **Server Response**: The server selects suitable cipher suites and compression methods from the client's list and responds with its public key and certificate in KEY\_EXCHANGE respectively.
* **Client Key Exchange**: The client responds with its public key in a KEY\_EXCHANGE message.
* **Certificate Verification**: Both parties verify the received certificates.
* **Finalization**: If verification is successful, both parties send a FINALIZATION message, marking the establishment of a secure channel.

# Security Considerations

* **Encryption**: Messages flagged as encrypted should be encrypted using AES-256.
* **Key Exchange**: Private/public key are generated using ECC algorithm.
* **Authentication**: Initial connection setup should include a handshake mechanism for authentication using digital signatures.

Note: - Authentication will be like key exchange the following are the steps involved

* 1. Server sends the certificate with signed by CA to Client which will be encrypted using key exchanged already.
  2. Client decrypts the certificate and checks the trusted authority for verification and connects to the CA to obtain the CA public key.
  3. Now the Client decrypts the secure encrypted public key in the certificate and verifies if the new public key matches the decrypted key authenticating the server.

# 5. Future Work

* Decoupling the existing payload data during handshake so that each stage will have its own packet of data.
* Verifying Certificate and allowing the client and server to have an option of creating a new AES key after the certificate is being transferred and verified.
* Allowing more Cipher Suites to supported by the client and server.
* And a layer of compression methods to help decrease the payload size during communication.