Secure Wireless Communication in Resource-Constrained Devices

Harsh Sarkar
B.Sc Semester VI, 2025
Visva-Bharati University

Abstract

Introduction

Traditional web communication relies heavily on **HTTP** and **HTTPS**.
HTTP (HyperText Transfer Protocol) is a simple and lightweight protocol widely used for data exchange. However, it lacks built-in security features, making it vulnerable to attacks such as **eavesdropping, spoofing**, and **data manipulation**.

HTTPS (HTTP Secure) addresses these vulnerabilities by adding a cryptographic layer using **SSL/TLS**, which provides **encryption**, **authentication**, and **data integrity**. While HTTPS is robust and widely adopted, its use in **resource-constrained devices** is often impractical. The **TLS handshake** process involves **computationally intensive operations** like public key cryptography and certificate validation, which impose high memory and processing overhead—exceeding the capabilities of lightweight embedded systems.

This project explores **alternative approaches to secure communication** that retain the simplicity of HTTP while incorporating essential security features. Specifically, it investigates how **HTTP can be modified or extended** to provide security without incurring the full overhead of HTTPS.

The study also evaluates the **trade-offs between security strength and resource consumption**, aiming to strike an optimal balance suitable for constrained devices.

By proposing **practical, low-overhead enhancements to HTTP**, this project contributes to the development of secure and efficient communication protocols for **embedded systems and IoT networks**, thereby enabling broader deployment of **trustworthy wireless technologies** in constrained environments.

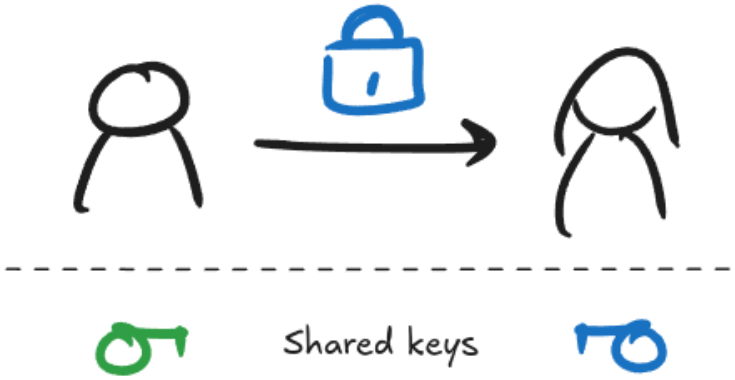
Background

Cryptography can be broadly categorized into two types:

1. Symmetric (secret key) cryptography
2. Asymmetric (public key) cryptography

Symmetric cryptography

In symmetric cryptography, the **same key is used for both encryption and decryption**. This means that both the sender and the receiver (and no one else) must possess the same secret key.



AES (Advanced Encryption Standard) is the current industry standard for symmetric cryptography. Although highly secure and widely used, it **requires a significant amount of memory and processing power**, mainly due to its complex substitution and permutation operations (S-boxes and P-boxes). This makes it **less suitable for constrained environments** like microcontrollers and IoT devices.

To address this, **ASCON** has been proposed as a lightweight alternative. ASCON is one of the **NIST-recommended algorithms** for lightweight authenticated encryption and hashing. It is specifically designed to be **simple, efficient, and suitable for resource-constrained devices**, making it a strong candidate for secure embedded applications. We will be using **ASCON** in this project.

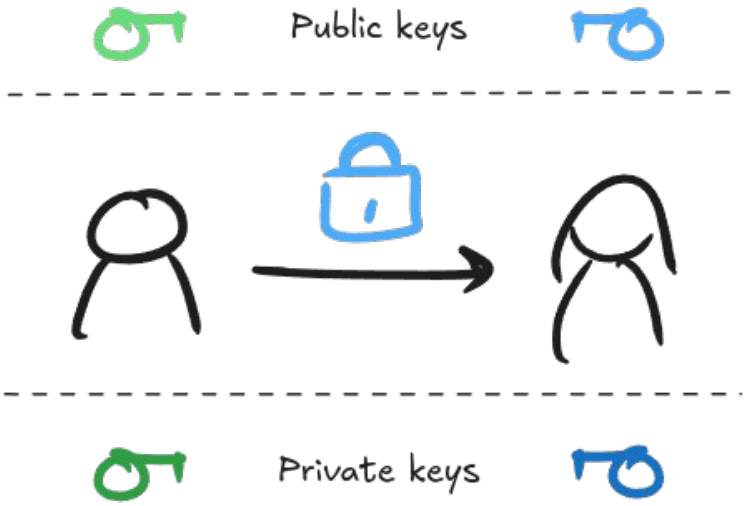
Before asymmetric cryptography was developed, keys had to be **shared physically or through secure channels**. This approach introduced several limitations:

- **Logistical issues:** Physically sharing keys is impractical, especially for large-scale or global communication networks.
- **Security risks:** Keys could be intercepted, copied, or stolen during transmission or transport.

- **Scalability problems**: In a network with n users, a unique key would be needed for every pair of users, resulting in $\frac{n(n-1)}{2}$ keys.

Asymmetric cryptography

In asymmetric cryptography, each person has a public and a private key. The **public key** is visible to everyone, but the **private key** is only known to the owner—not even the receiver knows the sender's private key. If Bob has to send a message to Alice, he encrypts the message with **Alice's public key**. The magic lies in the key generation process: keys are generated in such a way that the message can only be decrypted using **Alice's private key**.



RSA and **Diffie–Hellman** are the most commonly used asymmetric algorithms. They overcome the limitations of symmetric algorithms but are computationally expensive. The general approach is to **share a key using asymmetric techniques** and then use that key with symmetric algorithms for the actual data encryption. In this case, the asymmetric algorithm is also referred to as a **key-exchange algorithm**.

To protect against brute-force attacks, these algorithms require **very large key sizes** (in the order of thousands of bits!), and the computations involving these keys are also complex. Embedded devices and microcontrollers have **very limited memory and processing power**, so traditional asymmetric algorithms are often infeasible.

The alternative we shall be exploring is **Elliptic Curve Diffie–Hellman (ECDH)**. ECDH requires much **smaller key sizes** (for example, **256-bit ECDH offers security equivalent to 3072-bit RSA**) and hence performs significantly better in embedded systems.

ASCON cipher

Introduction

ASCON is a family of lightweight cryptographic algorithms designed for high performance in resource-constrained environments such as IoT, embedded systems, and hardware. It was selected as the **primary choice for lightweight authenticated encryption** by the NIST Lightweight Cryptography competition in 2023.

The ASCON family includes several variants tailored for different cryptographic needs:

- **ASCON-128** and **ASCON-128a**: Provide authenticated encryption with associated data (AEAD), suitable for general-purpose lightweight security. ASCON-128 offers a good balance of speed and security, while ASCON-128a is optimized for higher throughput.
- **ASCON-80pq**: A post-quantum variant offering stronger resistance to potential quantum attacks, with a 160-bit key and 128-bit security against quantum adversaries.
- **ASCON-HASH** and **ASCON-HASHA**: Lightweight cryptographic hash functions derived from the same permutation structure, suitable for hashing tasks in constrained environments.
- **ASCON-XOF**: An extendable-output function (XOF) for applications needing variable-length output, similar to SHAKE in the SHA-3 family.

ASCON-128

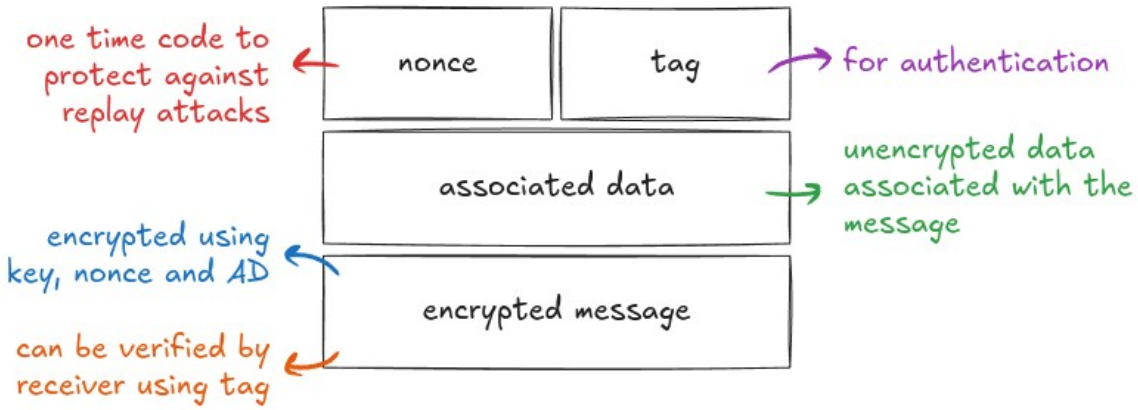
AEAD stands for **Authenticated Encryption with Associated Data**. It is a modern cryptographic technique that provides **both confidentiality and authenticity** in a single, unified operation.

Confidentiality ensures that the message (plain text) is encrypted and cannot be read by anyone without the correct key.

Authenticity guarantees that the message has not been tampered with and actually comes from the expected sender.

Usually, **encryption algorithms** are used for confidentiality and **hashing algorithms** are used for authenticity. ASCON-128 generates a 128-bit **tag** during the encryption process which can be used for authenticating the message.

Associated data (AD) refers to extra information (like headers, timestamps, or protocol metadata) that must be authenticated **but not encrypted**. This data is part of the integrity check but is transmitted in the clear.



Proposed approach

Implementation

Types

State: ASCON-128 uses a 320-bit state. The state is divided into five 64-bit words:

$$S = X_0 \parallel X_1 \parallel X_2 \parallel X_3 \parallel X_4$$

It can be represented as an array of five uint64's.

```
typedef uint64 State[5];
```

Key: ASCON-128 uses a 128-bit key, tag and nonce. They can be represented as an array of two uint64's.

```
typedef uint64 Key[2];
```

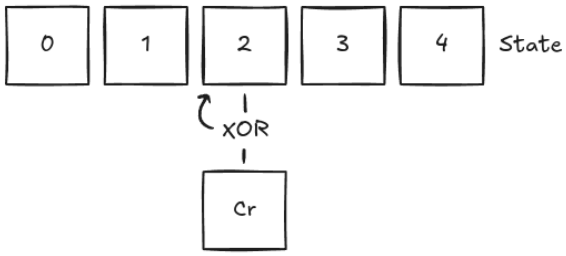
Stages of ASCON-128

Permutation

In ASCON-128, permutation consists of 3 operations.

1. **Round constant addition**

   Each round starts by XOR-ing a round-specific constant into `state[2]`.

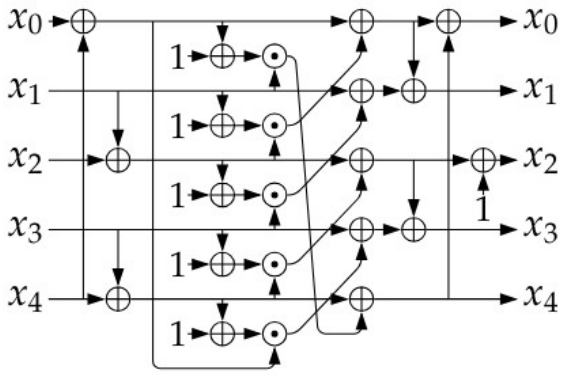



```
byte cr = 0xf0 - (12-rounds) * 0x0f;

for (int i = 0; i < rounds; ++i) {
   state[2] ^= cr;
   cr -= 0x0f;
}
```

2. **Substitution**

   A 5-bit non-linear substitution is applied to each 64-bit word using AND, XOR, and NOT logic.




```
void substitutionLayer(State state) {
   state[0] ^= state[4];
   state[4] ^= state[3];
   state[2] ^= state[1];
   for (int j = 0; j < 5; ++j) {
      state[j] ^= ~state[(j+1)%5] & state[(j+2)%5];
   }
   state[0] ^= state[4];
   state[1] ^= state[0];
   state[2] = ~state[2];
   state[3] ^= state[2];
}
```

3. **Linear diffusion**

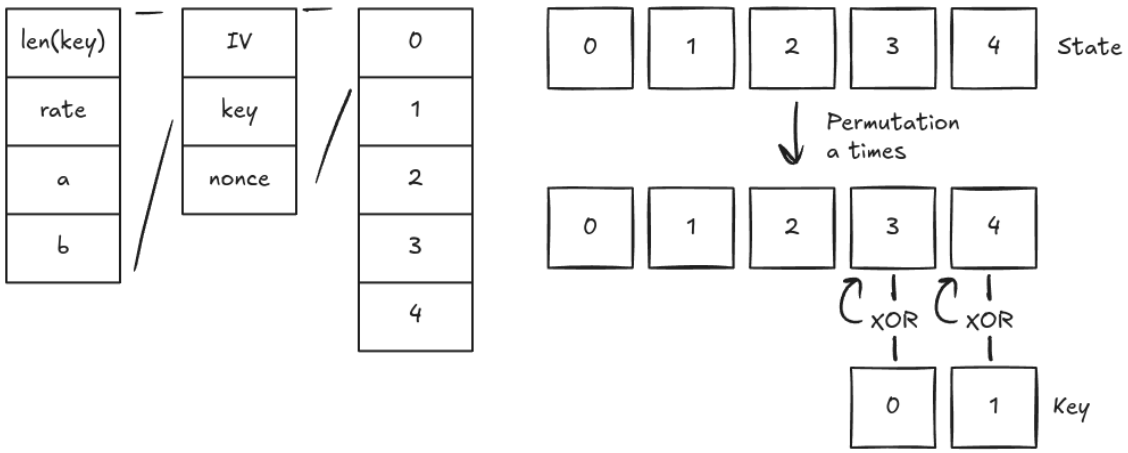
   Each state word is mixed by rotating and XOR-ing bits to increase diffusion.

$$
\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\
x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg \ 1) \oplus (x_2 \ggg \ 6) \\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg \ 7) \oplus (x_4 \ggg 41)
\end{aligned}
$$

```
void diffusionLayer(State state) {
   state[0] ^= crotr(state[0], 19) ^ crotr(state[0], 28);
   state[1] ^= crotr(state[1], 61) ^ crotr(state[1], 39);
   state[2] ^= crotr(state[2], 1) ^ crotr(state[2], 6);
   state[3] ^= crotr(state[3], 10) ^ crotr(state[3], 17);
   state[4] ^= crotr(state[4], 7) ^ crotr(state[4], 41);
}
```

Initialization

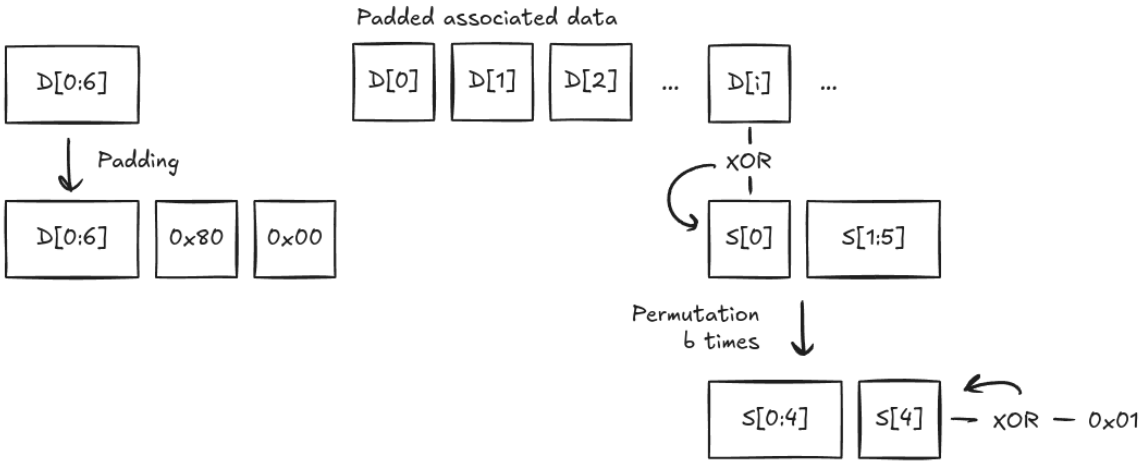
Initializes the state with a predefined IV, the key, and nonce. Then applies the permutation and mixes in the key again.




```
void asconInit(State state, Key key, Key nonce,
byte a, byte b) {
      uint64 IV = (uint64)KEYLEN << 56 | (uint64)(8 * RATE) << 48 |
            (uint64)a << 40 | (uint64)b << 32;
      state[0] = IV;
      state[1] = key[0];
      state[2] = key[1];
      state[3] = nonce[0];
      state[4] = nonce[1];
      asconPerm(state, a);
      state[3] ^= key[0];
      state[4] ^= key[1];
}
```

Processing associated data

If associated data is present, it is absorbed into the state block by block. First, the data is padded with a bit sequence of 10^*. Then each block is XOR-ed into `state[0]`, followed by a permutation. Finally, a domain separation marker (0x01) is XOR-ed at the end.

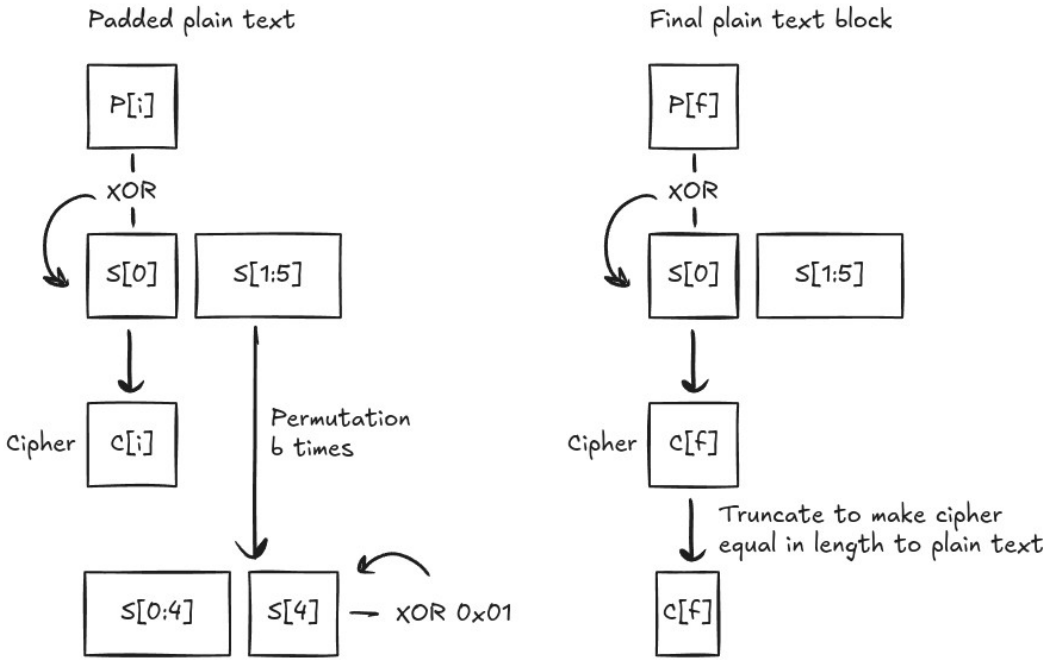



```
void asconProcessData(State state, byte b, ByteArr data) {
      if (data.length > 0) {
            ByteArr padded = getPaddedByteArr(data);
            for (int i = 0; i < padded.length/RATE; ++i) {
                  state[0] ^= padded.w[i];
                  asconPerm(state, b);
```

```
            }
            deleteByteArr(&padded);
        }
        state[4] ^= 0x01;
}
```

Processing plain text

Similar to associated data, the plaintext is first padded. Each plain text block is XOR-ed into `state[0]`, and the resulting value becomes the corresponding ciphertext block. After processing each block (except the last one), the state undergoes a `b`-round permutation. Finally, the cipher text is truncated to match the original length of the plain text.




```
ByteArr asconProcessText(State state, byte b, ByteArr plain) {
        ByteArr padded = getPaddedByteArr(plain);
        ByteArr cipher = createByteArr(padded.length);
        int i;
        for (i = 0; i < padded.length/RATE - 1; ++i) {
            state[0] ^= padded.w[i];
            cipher.w[i] = state[0];
            asconPerm(state, b);
        }
        state[0] ^= padded.w[i];
        cipher.w[i] = state[0];
        deleteByteArr(&padded);
        cipher.length = plain.length;
        resizeByteArr(&cipher, plain.length);
        return cipher;
}
```

Finalization

In the final stage, the key is reintroduced into the state. The permutation is applied again, and the key is XOR-ed a second time. The authentication tag is then extracted from the state to ensure integrity and authenticity.

```
void finalization(State state, Key key, byte a, Key tag) {
      state[1] ^= key[0];
      state[2] ^= key[1];
      asconPerm(state, a);
      tag[0] = state[3] ^ key[0];
      tag[1] = state[4] ^ key[1];
}
```

Evaluation

Conclusion

References