PROJECT No. ISNE P706-1/2568

Design and Implementation of a Lightweight Secure Communication System for IoT Using LoRa and MicroPython

Ritthanupahp Sitthananun	650615030
Piyawut Buncharoen	650615024
Siripa Aungwattana	650615032

A Report Submitted in Partial Fulfillment of Project Survey Course
as Required by the Degree of Bachelor of Engineering
Department of Computer Engineering
Faculty of Engineering
Chiang Mai University
2025

Project Title	: Design and Implementation of a Lightweight Secure Communication System for IoT Using LoRa and MicroPython
Name	: Ritthanupahp Sitthananun 650615030
	Piyawut Buncharoen 650615024
	Siripa Aungwattana 650615032
Department	: Computer Engineering
Project Advisor	: Asst. Prof. Kampol Woradit, Ph.D.
Degree	: Bachelor of Engineering
Program	: Information Systems and Network Engineering
Academic Year	: 2025

The Department of Computer Engineering, Faculty of Engineering, Chiang Mai University has approved this project to be part of the degree of Bachelor of Engineering (Information Systems and Network Engineering)

. . . . Department chair
(Assoc. Prof. Santi Phithakkitnukoon, Ph.D.)

Project examination committee:

. . . . Main advisor / Chair
(Asst. Prof. Kampol Woradit, Ph.D.)

. . . . Committee member
(Assoc. Prof. Anya Apavatjrut, Ph.D.)

. . . . Committee member
(Sasin Janpuangtong, Ph.D.)

Contents

Chapter 1
Introduction

1.1 Project Rationale

LoRa (Long Range) is a low-power wireless communication technology widely used in Internet of Things (IoT) applications such as environmental monitoring, infrastructure systems, and smart home appliances. Despite its advantages, LoRa lacks strong built-in security mechanisms. This leaves signals exposed to attacks such as eavesdropping, replay attacks, and message injections [1], [2]. Traditional encryption methods can mitigate these risks but are often too resource-intensive for low-power devices like the ESP32.

1.2 Objectives

To address this challenge, this project avoids static, pre-stored keys and instead generates encryption keys that are dynamic and constantly changing. RSSI (Received Signal Strength Indicator) values are used as the basis for key generation. Since RSSI values are unique between two devices and vary with environmental conditions, the resulting keys are never fixed. Unlike a static key, RSSI-based keys change over time, making them more difficult for an attacker to predict or reuse.

This project presents a lightweight secure communication framework for IoT systems using LoRa and MicroPython, focusing on encryption and dynamic key management. Session keys are updated using RSSI values. Both devices share a predefined keyword and an RSSI interval for adjustment. The communication process begins with the sender transmitting a verification message while the receiver measures the RSSI and iteratively adjusts its interpretation within the agreed interval until the message is correctly decrypted and the keyword matches. This ensures that both devices converge on the same session key despite natural fluctuations in signal strength. Furthermore, by using RSSI values with two-decimal precision and combining measurements across multiple LoRa channels, the system significantly increases resistance to interception and eavesdropping. Overall, the framework provides a practical balance between security and performance for securing IoT communication [3],[4].

1.3 Project Scope

The scope of this project focuses on designing and developing a secure communication framework based on LoRa technology using MicroPython for low-power IoT devices such as the ESP32. The project emphasizes generating dynamic encryption keys from signal strength values (RSSI) to enhance communication security, as well as conducting communication experiments between at least two devices.

1.4 Expected Outcomes

1.4.1 Secure IoT Communication Prototype

A working prototype of a lightweight secure communication framework implemented on ESP32 boards with LoRa modules, demonstrating reliable data exchange using MicroPython.

1.4.2 Dynamic Key Generation Mechanism

Successful implementation of RSSI-based session key generation, ensuring that encryption keys are dynamic, unpredictable, and resistant to reuse by attackers.

1.5 Technology and Tools

1.5.1 Hardware Technology

The hardware used in this project consists of the following components:

- **ESP32 Development Board**
  - Serves as the main microcontroller for implementing the secure communication framework.
  - Provides sufficient processing power, memory, and Wi-Fi/Bluetooth capabilities while maintaining low energy consumption.
  - Runs MicroPython firmware for ease of prototyping and testing.
- **LoRa Transceiver Module**
  - Enables long-range, low-power wireless communication between IoT devices.
  - Used to transmit and receive data packets.
- **Jumper Wires and Breadboard**
  - Used to connect ESP32 boards to LoRa modules for prototyping purposes.
  - Provides flexibility for testing different circuit configurations in a laboratory environment.
- **Power Supply (USB)**
  - Each ESP32 board is powered via USB from a computer or adapter.
  - Provides stable voltage and allows simultaneous programming and monitoring.

1.5.2 Software Technology

- **MicroPython**
  - Lightweight Python implementation designed for microcontrollers such as ESP32.
  - Provides built-in support for hardware interfaces (GPIO, SPI, UART) and efficient execution on resource-constrained devices.
  - Used to implement the encryption, decryption, and RSSI-based key generation mechanisms.
- **Visual Studio Code (VS Code)**
  - Serves as the primary code editor and development environment.
  - Offers extensions for MicroPython, debugging, and serial communication with ESP32 devices.
- **Thonny IDE**
  - Beginner-friendly IDE optimized for MicroPython programming.
  - Used for quick prototyping, uploading scripts, and monitoring device outputs.
- **Mpremote (MicroPython Remote Tool)**
  - Command-line utility for interacting with MicroPython devices over USB.
  - Supports file transfer, script execution, and device management.
- **Serial Monitor Tools**
  - Used to display and debug communication between ESP32 devices in real time.
  - Allows monitoring of RSSI values, encryption/decryption outputs, and keyword verification results.

1.6 Project Plan

Task	Jun 2025	Jul 2025	Aug 2025	Sep 2025	Oct 2025
Project Discussion	■	■			
Requirement Analysis		■	■		
System Design			■		
Prototype Implementation			■	■	

Task	Jun 2025	Jul 2025	Aug 2025	Sep 2025	Oct 2025
Presentation				▨	▨
Final Report Documentation					▨

1.7 Roles and Responsibilities

This project is made possible by 3 students and 1 advisor.

- Ritthanupahp Sitthananun: Responsible for coding and demo.
- Piyuwut Buncharoen: Responsible for implementing hardware.
- Siripa Aungwattana: Responsible for implementing hardware.
- Kampol Woradit: Advisor providing suggestions and support.

1.8 Impacts of This Project on Society, Health, Safety, Legal, and Cultural Issues

This project focuses on developing secure and reliable IoT communication. Users can transmit and receive data with a reduced risk of leakage or attacks. The system is low-power, minimizing health impacts, and contains no illegal or inappropriate content or functions. Therefore, it does not cause significant adverse effects on society or culture.

Chapter 2
Background Knowledge and Theory

2.1 Internet of Things (IoT)

The Internet of Things (IoT) refers to a network of interconnected devices that can sense, process, and exchange data with minimal human intervention. These devices are widely deployed in applications such as smart homes, healthcare monitoring, environmental sensing, transportation systems, and industrial automation. While IoT enables efficiency and automation, it also introduces security challenges. Many IoT devices are resource-constrained in terms of processing power, memory, and energy supply, making it difficult to implement traditional, computation-heavy cryptographic techniques. As a result, IoT networks are often vulnerable to attacks such as eavesdropping, replay, and message injection.

2.2 LoRa Technology

LoRa (Long Range) is a low-power wide-area network (LPWAN) communication technology that enables devices to transmit data over several kilometers while consuming minimal energy. This makes it suitable for IoT applications where devices must operate on battery power for extended periods. LoRa achieves long-distance communication using Chirp Spread Spectrum (CSS) modulation, which provides robustness against noise and interference. However, LoRa's primary limitation is its lack of built-in security mechanisms. The physical layer itself does not provide strong confidentiality or integrity protection. While LoRaWAN (the higher-layer protocol) introduces some security features, lightweight LoRa implementations—such as those used in simple ESP32 + LoRa projects—are highly vulnerable to interception, spoofing, and key extraction if insecure practices (e.g., hardcoded keys) are used.

2.3 Received Signal Strength Indicator (RSSI)

The Received Signal Strength Indicator (RSSI) measures the power level of a received wireless signal, typically expressed in decibels relative to one milliwatt (dBm). In LoRa systems, RSSI is automatically measured at the receiver whenever a packet is received. Normally, RSSI is used to evaluate link quality or assist in adaptive communication strategies. In the context of security, RSSI can also be leveraged as a source of entropy for key generation. Since RSSI values fluctuate depending on distance, obstacles, interference, and environmental conditions, they are inherently dynamic and difficult for an attacker to predict without being physically co-located in the communication channel. By using RSSI values as a basis for session key generation, IoT devices can avoid reliance on static, pre-shared keys that are easy to compromise.

2.4 Lightweight Cryptography

Lightweight cryptography refers to cryptographic techniques designed specifically for devices with limited computational and memory resources. Unlike traditional algorithms such as AES-256 or RSA, which require significant processing power, lightweight algorithms are optimized for efficiency while maintaining an acceptable level of security. Typical strategies include reducing key sizes, minimizing memory overhead, or designing algorithms tailored for 8-bit or 32-bit microcontrollers. For IoT applications using devices like the ESP32, lightweight cryptography is essential to balance security with system performance. The challenge is to implement schemes that protect data confidentiality and integrity without exceeding constraints such as 10 KB of program memory or 300 bytes of RAM.

2.5 Key Management in IoT

Key management is one of the most critical aspects of securing IoT networks. Traditional approaches rely on pre-shared static keys, which pose significant risks: once an attacker obtains the key, they can decrypt all subsequent communications. Dynamic key management schemes provide stronger protection by regularly updating session keys. One promising approach is to derive session keys from physical-layer metrics, such as RSSI. Since each device independently measures RSSI during communication, keys can be generated locally without transmitting sensitive information over the air. To ensure synchronization, both devices agree on an adjustment interval and use a shared keyword for verification. This method significantly reduces the risk of interception and makes it difficult for attackers to reuse keys, thereby enhancing overall IoT security.

Chapter 3
Project Structure

3.1 Methodology

3.1.1 System Architecture

The proposed framework consists of two ESP32 devices, each connected to a LoRa SX1276 module. One device operates as the sender (initiator), and the other as the receiver (responder). The devices communicate over LoRa to exchange verification and application messages. The overall architecture ensures that both devices can generate identical session keys without directly transmitting key material.

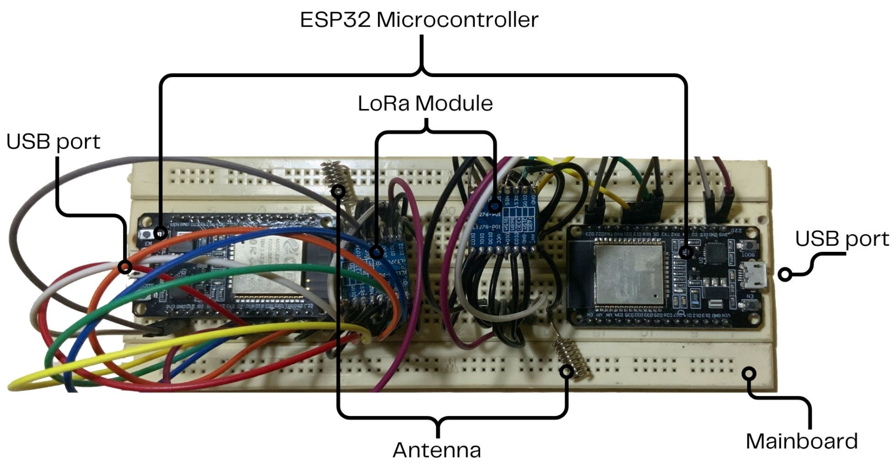

Figure 3.1: Design Architecture

3.1.2 RSSI-Based Key Generation

To synchronize RSSI values, one device (the initiator) first transmits a known verification message. The receiving device measures the RSSI of this signal and attempts to decode the message using its reading. If decryption fails, the receiver iteratively adjusts its interpretation by shifting the RSSI value up or down within a pre-agreed tolerance interval (e.g., ±0.5 dBm) until the message is correctly decrypted. Unlike static pre-shared keys, which remain fixed and are vulnerable once compromised, session keys in this framework are dynamically derived from RSSI. Since RSSI is measured uniquely at the receiver and varies naturally with the environment, it is difficult for an attacker to predict or replicate. Security is further enhanced by concatenating the RSSI values and sampling them across multiple frequency channels into an array.

3.1.3 Keyword Verification

To ensure correctness, both devices share a pre-defined keyword. After decryption, the receiver compares the result against this keyword. A match confirms that both devices are synchronized on the same session key. If the keyword is not matched, the process repeats until a valid session key is established.

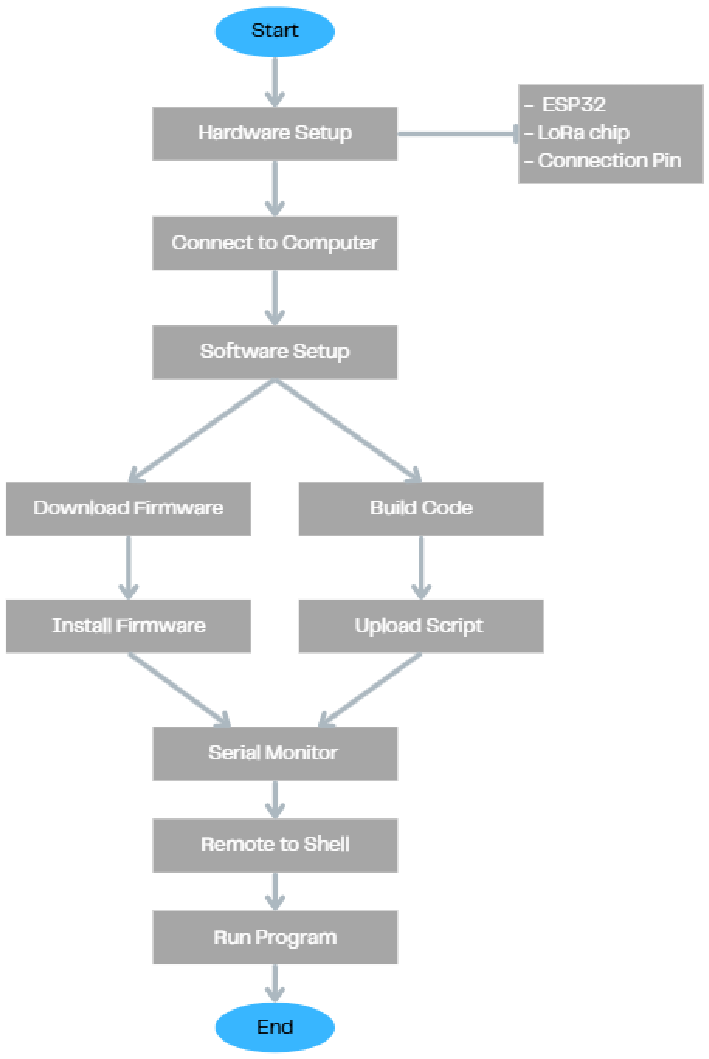

Figure 3.2: Implementation Flow

3.1.4 System Implementation

Hardware Setup

The following components are required for the system implementation:

- ESP32 development board (MicroPython-compatible)

- SX1276 LoRa module
- USB cable for ESP32
- Jumper wires
- Computer with `esptool.py` and `mpremote` installed

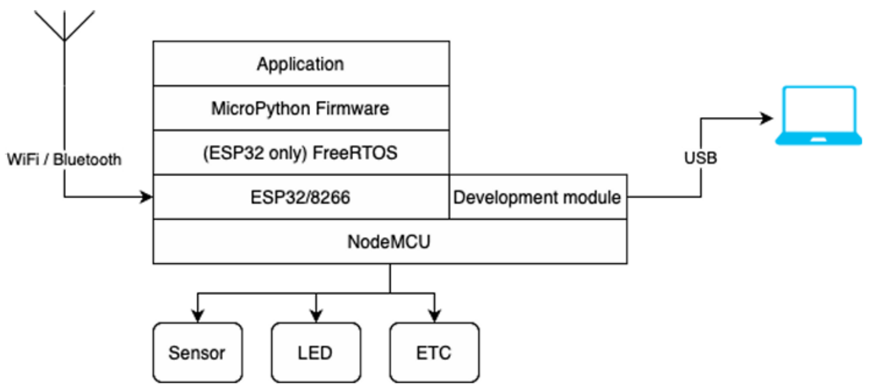

Figure 3.3: Module System

Pin Mapping

The interfacing between the ESP32 and the SX1276 module is established through the SPI bus and control lines. The pin assignments are listed in Table 3.1.4. These may be modified in `lora.py` as required.

Signal	**ESP32 GPIO Pin**
MISO	GPIO19
MOSI	GPIO23
SCK	GPIO18
CS	GPIO5
RST	GPIO17
DIO0	GPIO26

Table 3.1: Pin Mapping between ESP32 and LoRa Module

3.1.5 Flashing MicroPython on ESP32

To run the ESP32 with MicroPython, follow these steps:

Erase Existing Flash

Before flashing, erase the current firmware on the ESP32:

```
esptool --port COM6 erase_flash
```

This clears the flash memory of a standard ESP32.

Write MicroPython Firmware

Next, write the MicroPython firmware to the ESP32. Replace <micropython-firmware.bin> with your firmware file name:

```
esptool --baud 460800 write_flash 0 \
ESP32_GENERIC_C6-20250415-v1.25.0.bin
```

Example using a specific firmware file:

```
esptool --baud 460800 write_flash 0 ESP32_GENERIC_C6-20250415-v1.25.0.bin
```

After flashing, the ESP32 is ready to run MicroPython scripts via a serial connection.

3.1.6 Prepare and Upload Files

Before running the system, edit the `lora.py` file locally to ensure the pin mapping matches your ESP32 configuration (see Section 3.1.4).

Upload `lora.py` to ESP32

Use `mpremote` to upload the Python script to the ESP32:

```
mpremote connect COM6 fs cp lora.py
```

Explanation:

- `mpremote connect COM6` — Connects to the ESP32 on port COM6.
- `fs cp lora.py :` — Copies the `lora.py` file to the root directory of the ESP32 filesystem.

After uploading, the ESP32 is ready to run the LoRa communication program.

3.1.7 Connect to Shell and Run

After uploading the necessary files, follow these steps to execute the program on the ESP32 nodes.

1. Connect the ESP32 to your computer via USB.
2. Open the serial REPL using `mpremote`:

   ```
   mpremote connect /dev/ttyUSB0 repl
   ```

3. Upload and run the application scripts:

- On Node A (sender): upload and run `sender.py`.
- On Node B (receiver): upload and run `receiver.py`.

Once both nodes are running, they can communicate over LoRa, exchange verification messages, and generate session keys dynamically.

3.1.8 LoRa Communication Script

The following Python script (`sender.py`) demonstrates sending and receiving LoRa packets using the SX1276 module on an ESP32. It includes unicast requests, broadcast messages, and handling of packet reception.

3.1.9 LoRa Communication Script

The following Python script (`sender.py`) demonstrates sending and receiving LoRa packets using the SX1276 module on an ESP32. It includes unicast requests, broadcast messages, and handling of packet reception.

```
from machine import Pin
import time, urandom as random
from lora import SX1276

# Pin configuration
LoRa_MISO_Pin = 19
LoRa_MOSI_Pin = 23
LoRa_SCK_Pin  = 18
LoRa_CS_Pin   = 5
LoRa_RST_Pin  = 17
LoRa_DIO0_Pin = 27
LoRa_DIO1_Pin = 35
LoRa_DIO2_Pin = 34
SPI_CH        = 2  # VSPI

# Random channel hopping setup
random.seed(11)
channels2Hopping = [914_000_000 + 200_000 * random.randint(0, 10) for i in range(128)]  # 914~916 MHz

LoRa_id = 1
lora = SX1276(LoRa_RST_Pin, LoRa_CS_Pin, SPI_CH, LoRa_SCK_Pin, LoRa_MOSI_Pin, LoRa_MISO_Pin,
              LoRa_DIO0_Pin, LoRa_DIO1_Pin, LoRa_id, channels2Hopping, debug=False)

# Packet reception handler
def get_payload(self, data, SNR, RSSI):
    global received_payload
    received_payload = data

lora.req_packet_handler = get_payload

###########################################
# First REQ packet
###########################################
payload = str(random.randint(100,65536)) + ") Hello~"
print('[Sending]', payload)
```

```
lora.send(dst_id=0, msg=payload, pkt_type=lora.PKT_TYPE['REQ'])
while not lora.is_available:
    time.sleep(1)

###########################################
# Two-way communication (receive)
###########################################
received_payload = None
lora.mode = 'RXCONTINUOUS'
while not lora.is_available:
    time.sleep(1)
print('[Received] What we receive from the receiver is:', received_payload)

###########################################
# Send REQ packet to wrong receiver
###########################################
payload = str(random.randint(100,65536)) + ") You will not receive this packet because we specified a wrong dst_id"
print('[Sending]', payload)
lora.send(dst_id=3, msg=payload, pkt_type=lora.PKT_TYPE['REQ'], timeout=10, retry=3, debug=True)

for i in range(10):
    if lora.is_available: break
    time.sleep(1)
else:
    print("[Note] you will see this line because lora.is_available is always false")

###########################################
# Send broadcast packets
###########################################
time.sleep(10)
payload = str(random.randint(100,65536)) + ") This long BRD packet will be received"
print('[Sending]', payload)
lora.send(dst_id=0, msg=payload, pkt_type=lora.PKT_TYPE['BRD'])

time.sleep(10)
payload = str(random.randint(100,65536)) + ") This long BRD packet will also be received
print('[Sending]', payload)
lora.send(dst_id=3, msg=payload, pkt_type=lora.PKT_TYPE['BRD'])
```

This script demonstrates:

- Sending a unicast request packet (`REQ`) to a specific node.
- Receiving packets and handling RSSI/SNR values.
- Handling cases where the destination ID is incorrect.
- Sending broadcast (`BRD`) packets that are received by all nodes, regardless of the destination ID.

3.1.10 Receiver Node Script

The following Python script (`receiver.py`) demonstrates the ESP32 receiving LoRa packets, replying to unicast requests, and handling broadcast packets.

```
from machine import Pin
import time, urandom as random
from lora import SX1276

# Pin configuration
LoRa_MISO_Pin = 19
LoRa_MOSI_Pin = 23
LoRa_SCK_Pin  = 18
LoRa_CS_Pin   = 5
LoRa_RST_Pin  = 17
LoRa_DIO0_Pin = 27
LoRa_DIO1_Pin = 35
LoRa_DIO2_Pin = 34
SPI_CH        = 2  # VSPI

# Channel hopping setup (both sender and receiver must know the sequence)
random.seed(11)
channels2Hopping = [914_000_000 + 200_000 * random.randint(0,10) for i in range(128)]

LoRa_id = 0
lora = SX1276(LoRa_RST_Pin, LoRa_CS_Pin, SPI_CH, LoRa_SCK_Pin, LoRa_MOSI_Pin, LoRa_MISO_Pin,
LoRa_DIO0_Pin, LoRa_DIO1_Pin, LoRa_id, channels2Hopping, debug=False)

# Packet handlers
def get_payload(self, data, SNR, RSSI):
    global received_payload
    received_payload = data

lora.req_packet_handler = get_payload
lora.brd_packet_handler = lambda self, data, SNR, RSSI: print("[BRD]", data)

###########################################
# Prepare to receive first REQ packet
###########################################
received_payload = None
lora.mode = 'RXCONTINUOUS'
while not lora.is_available:
    time.sleep(1)

print("[Note] We will see this line after receiver ACKed the first REQ packet with an ACK packet. "
      "Then the receiver becomes the new sender.")

# Reply to 'Hello~' packet
print('[Received]', received_payload)
if received_payload[-6:] != b'Hello~': raise

payload = str(random.randint(100,65536)) + ") Hi ~ I have received your hello"
lora.send(dst_id=1, msg=payload, pkt_type=lora.PKT_TYPE['REQ'])
print('[Sending]', payload)
while not lora.is_available:  # Stop if our reply got acknowledged
    time.sleep(1)

###########################################
# Prepare to receive two BRD packets
###########################################
received_payload = None
```

```
lora.mode = 'RXCONTINUOUS'
while not lora.is_available:
    time.sleep(1)

print("[Note] This line will not be reached because BRD is not two-way communication. "
      "The receiver will not stop listening.")
```

- Continuous reception of REQ packets.
- Replying to a unicast request (two-way communication).
- Handling broadcast (BRD) packets, which are received but do not trigger acknowledgment.

Chapter 4
System Evaluation

4.1 Evaluation Metrics

If implemented, the proposed framework would be evaluated using the following criteria:

4.1.1 Performance

the time required for key generation, encryption, and decryption on ESP32 devices.

4.1.2 Resource Usage

memory and CPU consumption, ensuring the design remains lightweight.

4.1.3 Security Assessment

resistance against common IoT attacks, including replay and man-in-the-middle (MITM) attacks.

4.2 Evaluation Method

4.2.1 Correctness Testing

Repeatedly transmit verification messages and record the percentage of times the receiver derives the correct key and matches the predefined keyword.

4.2.2 Performance Measurement

Measure execution time of key generation and encryption/decryption functions using built-in timers on ESP32.

4.2.3 Resource Usage

Monitor program memory footprint and RAM usage to ensure they remain under the target limits (10 KB flash, 300 bytes RAM).

4.2.4 Security Testing

Simulate potential attack scenarios such as replaying old packets or attempting to intercept communication, to evaluate whether the dynamic key mechanism prevents message reuse or prediction.

4.3 Evaluation Method

The evaluation is expected to show that the RSSI-based key generation approach achieves high correctness in key agreement, maintains lightweight resource usage suitable for ESP32 devices, and provides improved resilience against interception compared to static key methods.

4.4 Comparative Discussion with Related Works

To assess the contribution of the proposed RSSI-based lightweight secure communication framework, it is useful to compare it with recent research in LoRa security.

Chaos and Timing Approaches

Erkan et al. (2023) proposed a chaos-based encryption with a timing confirmation algorithm on LoRa. They use a Logistic map as a pseudo-random generator and millisecond-level timing signals for message validation. Their evaluation, including NIST randomness and entropy tests, shows that chaotic keys provide high unpredictability. This study, however, focuses on theoretical analysis of randomness and timing rather than memory or CPU usage constraints [3].

Lightweight Block Cipher on ESP32

Lim et al. (2024) designed a lightweight block cipher for ESP32 based on AES/DES principles. The algorithm uses custom S-boxes and P-boxes for encryption and is validated through SAC, monobit, and correlation tests. It demonstrates feasibility on ESP32 with approximately 3 ms execution time per encryption and randomness performance close to optimal. However, it depends on static pre-shared keys, increasing complexity in key management [4].

Frequency-Hopping Network Architectures

Ortigoso et al. (2024) introduced Dynamic Dual Frequency Hopping (DDFH), a network-level architecture using dual radios and mesh topology to improve scalability and mitigate eavesdropping. Their solution achieved a $\sim 94.66\%$ packet success rate in prototypes. While effective, DDFH targets large-scale LoRaWAN deployments with Raspberry Pi and dual-antenna hardware. In contrast, our approach is device-centric, requiring only low-cost ESP32 and LoRa modules [5].

Physical-Layer RSSI + CRT Encryption

Zhang et al. (2021) proposed a physical-layer scheme combining RSSI-based key extraction with the Chinese Remainder Theorem (CRT) to generate cyclic shift encryption factors.

Their algorithm preserves BER performance while degrading eavesdropper decoding to near 0.5 BER. Our work shares the principle of RSSI-based key selection but focuses more on iterative keyword verification for synchronization instead of CRT [6].

Advantages of the Proposed Work

- Does not require static key storage, unlike AES/DES-based methods.
- Avoids additional timing functions present in chaos-based approaches.
- Lightweight enough for ESP32 with MicroPython, unlike DDFH requiring dual radios/ SBCs.
- Implements practical verification (keyword matching) suitable for low-resource devices.

Limitations

- RSSI values are environment-dependent, which requires careful testing under varying conditions compared to chaos-based schemes.
- Keyword verification may be less complex than CRT-based physical-layer methods.
- Does not address large-scale network-level threats such as jamming, unlike frequency-hopping solutions.

References

[1] E. Aras, G. S. Ramachandran, P. Lawrence, and D. Hughes. Exploring the security vulnerabilities of lora. In *2017 3rd IEEE International Conference on Cybernetics (CYBCONF)*, pages 1–6, Exeter, UK, 2017.

[2] Semtech Corporation. SX1276/77/78/79 – 137 MHz to 1020 MHz Low Power Long Range Transceiver (Wireless, Sensing & Timing Datasheet), 2016. Rev. 5, Aug. 2016.

[3] E. Erkan, H. Oğraş, and Ş. Fidan. Application of a secure data transmission with an effective timing algorithm based on lora modulation and chaos. *Microprocessors and Microsystems*, 99:104829, 2023.

[4] L. C. Ni, S. Ali, A. N. A. A. Aziz, and R. A. Rashid. Implementation of proposed cryptography algorithm on esp32-based iot system. In *2024 IEEE International Conference on Advanced Telecommunication and Networking Technologies (ATNT)*, pages 1–4, Johor Bahru, Malaysia, 2024.

[5] A. R. Ortigoso, G. Vieira, D. Fuentes, L. Frazão, N. Costa, and A. Pereira. Ddfh: Dynamic dual frequency hopping for lora networks. In *2024 34th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–4, Sydney, Australia, 2024.

[6] C. Zhang, J. Yue, L. Jiao, J. Shi, and S. Wang. A novel physical layer encryption algorithm for lora. *IEEE Communications Letters*, 25(8):2512–2516, 2021.