

**SECURED AGRICULTURE SENSOR DATA BASED ON END-TO-END  
ENCRYPTION USING RASPBERRY PI**

**BY  
CHOO JIA HUEY**

**A REPORT  
SUBMITTED TO  
Universiti Tunku Abdul Rahman  
in partial fulfillment of the requirements  
for the degree of  
BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMMUNICATIONS  
AND NETWORKING  
Faculty of Information and Communication Technology  
(Kampar Campus)**

**FEBRUARY 2025**

## **COPYRIGHT STATEMENT**

© 2025 Choo Jia Huey. All rights reserved.

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Technology (Honours) Communications and Networking at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

## **ACKNOWLEDGEMENTS**

I would like to express my sincere thanks and appreciation to my supervisor, Dr. Abdulrahman Aminu Ghali who has given me this bright opportunity to engage in this project. It is my first step to establish a career in the agriculture security field. A million thanks to you.

To a very special person in my life, Low Hui Wen, for her patience, unconditional support, and love, and for standing by my side during hard times. Finally, I must say thanks to my parents and my family for their love, support, and continuous encouragement throughout the course.

## **ABSTRACT**

The rise of smart agriculture in Malaysia, powered by IoT sensor networks, has transformed farming by enabling real-time monitoring of soil moisture, temperature, and environmental conditions. However, much of this sensor data is transmitted without encryption, exposing it to risks such as interception, tampering, and unauthorised access. This project addresses these security concerns by developing a secure data collection and visualisation system using Raspberry Pi and Node-RED. The system integrates multiple robust encryption algorithms—AES-128, AES-256, ChaCha20, and Twofish—for end-to-end data protection. Additionally, a benchmarking tool was developed to evaluate and compare the performance of these algorithms in terms of speed, memory, CPU usage, and encryption overhead. The final outcome is a lightweight, replicable solution for secure smart agriculture systems that enhances trust, integrity, and data privacy in IoT-based farming environments.

Area of Study (Minimum 1 and Maximum 2): Internet of Things, Cybersecurity

Keywords (Minimum 5 and Maximum 10): Security, Smart Agriculture, Encrypted Data Transmission, Sensor Networks, Node-RED, MQTT, Raspberry Pi, Real-time Monitoring, ChaCha20, Twofish

# TABLE OF CONTENTS

<b>TITLE PAGE</b>	<b>i</b>
<b>COPYRIGHT STATEMENT</b>	<b>ii</b>
<b>ACKNOWLEDGEMENTS</b>	<b>iii</b>
<b>ABSTRACT</b>	<b>iv</b>
<b>TABLE OF CONTENTS</b>	<b>v</b>
<b>LIST OF FIGURES</b>	<b>viii</b>
<b>LIST OF TABLES</b>	<b>ix</b>
<b>LIST OF SYMBOLS</b>	<b>x</b>
<b>LIST OF ABBREVIATIONS</b>	<b>xi</b>
<b>CHAPTER 1 INTRODUCTION</b>	<b>1</b>
1.1 Problem Statement and Motivation	2
1.2 Objectives	2
1.3 Project Scope and Direction	3
1.4 Contributions	3
1.5 Report Organization	3
<b>CHAPTER 2 LITERATURE REVIEW</b>	<b>5</b>
2.1 Review of Technologies	4
2.1.1 Raspberry Pi	5
2.1.2 Node-RED	5
2.1.3 MQTT Protocol	5
2.1.4 Python Programming Language	6
2.1.5 Encryption Algorithms	6
2.1.6 Data Visualisation and Monitoring Tools	6
2.2 Related Works in Secure Smart Farming	7
2.2.1 Precision Agriculture Monitoring System using Wireless Sensor Network and Raspberry Pi Local Server	7
2.2.2 Cyber Attack on Smart Farming Infrastructure	9
2.2.3 Securing the Internet of Battlefield Things with ChaCha20	11

2.2.4	Comparative Performance Analysis of Lightweight Cryptography Algorithms	13
2.3	Limitations of Previous Study	14
2.4	Summary	16
<b>CHAPTER 3</b>	<b>SYSTEM METHODOLOGY</b>	<b>17</b>
3.1	System Design Diagram	17
3.1.1	System Architecture Diagram	17
3.1.2	Use Case Diagram and Description	18
3.1.3	Activity Diagram	19
3.2	Tools and Technologies Used	20
3.2.1	Hardware	20
3.2.2	Software Tools	20
3.2.3	Python Libraries and Packages	21
3.3	Secure Data Transmission Workflow	22
3.3.1	Encryption Mode Selection	22
3.3.2	Sensor Data Generation	22
3.3.3	MQTT Publishing	23
3.3.4	Encryption Logic	23
3.4	Node-RED Decryption and Visualisation Workflow	23
3.4.1	MQTT Message Reception	23
3.4.2	Decryption Flow	24
3.4.3	Dashboard Integration	24
3.5	Benchmarking and Methodology	24
3.5.1	Benchmarking Script Structure	25
3.5.2	Performance Metrics	25
3.5.3	Testing Parameters	25
3.5.4	Data Collection Tools	26
3.5.5	Algorithm Executions	26
3.5.6	Summary Results Output	26
3.6	Data Collection and Analysis Process	26
3.6.1	Structured Metric Recording	26
3.6.2	Tabulated Output	27

3.6.3	Visualisation and Export	27
3.6.4	Reproducibility	27
3.7	Summary	28
<b>CHAPTER 4</b>	<b>SYSTEM DESIGN</b>	<b>29</b>
4.1	System Block Diagram	29
4.2	System Components Specifications	30
4.2.1	Hardware Components	30
4.2.2	Software Components	30
4.2.3	Setup for Node-RED Function Global Context	31
4.3	Circuits and Components Design	32
4.3.1	Simulated Sensor Emulation	32
4.3.2	Potential Hardware Integration	32
4.4	System Components Interaction Operations	33
4.4.1	Sender Operations (Python Script)	33
4.4.2	Message Transmission (MQTT Broker)	33
4.4.3	Receiver Operations (Node-RED)	33
4.4.4	Dashboard Visualisation	34
<b>CHAPTER 5</b>	<b>SYSTEM IMPLEMENTATION</b>	<b>35</b>
5.1	Hardware Setup	35
5.2	Software Setup	36
5.3	Setting and Configuration	38
5.3.1	Encryption Script Configuration (Python)	38
5.3.2	Mosquitto MQTT Broker Setup	38
5.3.3	Node-RED Environment Configuration	39
5.3.4	Flow Import and Dashboard Setup	39
5.4	System Operation (with Screenshot)	39
5.4.1	Starting the Sender Script	40
5.4.2	MQTT Broker Operation (Mosquitto)	40
5.4.3	Receiving and Decrypting in Node-RED	41
5.4.4	Dashboard Visualisation	42
5.4.5	Benchmark Script Execution Flow	42

5.5	Implementation Issues and Challenges	43
5.5.1	Node-RED function Context Configuration	43
5.5.2	Block Cipher Padding Constraints	44
5.5.3	Complexity of Twofish Implementation in Node-RED	44
5.6	Concluding Remark	44
<b>CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION</b>		<b>45</b>
6.1	System Testing and Performance Metrics	45
6.1.1	Objectives of Testing	45
6.1.2	Test Parameters and Methodology	45
6.1.3	Entropy Conditions	46
6.1.4	Repetitions	46
6.1.5	Performance Metric Tracked	46
6.2	Testing Setup and Result	47
6.2.1	Encryption Time	47
6.2.2	Decryption Time	48
6.2.3	Encryption Throughput	49
6.2.4	Decryption Throughput	50
6.2.5	Memory Usage	52
6.2.6	CPU Usage	54
6.2.7	Energy Estimation	56
6.3	Project Challenges	57
6.3.1	Performance Bottlenecks with Twofish	57
6.3.2	Resource Constraints on the Raspberry Pi	58
6.3.3	Memory and Energy Measurement Anomalies	58
6.3.4	Library Fragmentation and Tool Limitations	58
6.3.5	Manual Interruption Handling for Long-running Tests	59
6.4	Objectives Evaluation	59
6.5	Concluding Remark	60
<b>CHAPTER 7 CONCLUSION AND RECOMMENDATION</b>		<b>61</b>
7.1	Conclusion	61



7.2 Recommendation	61
<b>REFERENCES</b>	<b>63</b>
<b>APPENDIX</b>	<b>65</b>
<b>POSTER</b>	<b>65</b>

# LIST OF FIGURES

Figure Number	Title	Page
Figure 2.1	System Architecture and Attack Vector	10
Figure 2.2	Graphical Depiction of a Deauthentication Attack	11
Figure 3.1	System Architecture Diagram	17
Figure 3.2	Use Case Diagram	18
Figure 3.3	Activity Diagram	19
Figure 4.1	System Block Diagram	29
Figure 5.1	Demonstration of Python Script	40
Figure 5.2	Verification of MQTT Broker Active	40
Figure 5.3	Node-RED Flow	41
Figure 5.4	Node-RED Dashboard UI View	42
Figure 5.5	Benchmarking Script Result Demonstration	43
Figure 6.1	Encryption Time Comparison Across Algorithms and Devices (Lower is Better)	47
Figure 6.2	Decryption Time Comparison Across Algorithms and Devices (Lower is Better)	48
Figure 6.3	Encryption Throughput Comparison Across Mac and Raspberry Pi (Higher is Better)	49
Figure 6.4	Decryption Throughput Comparison Across Mac and Raspberry Pi (Higher is Better)	51
Figure 6.5a	Memory Usage of Encryption Algorithms on Macbook	52
Figure 6.5b	Memory Usage of Encryption Algorithms on Raspberry Pi	53
Figure 6.6a	CPU Usage per Encryption/Decryption Operation at 1MB on Mac and Pi (Lower is Better)	54
Figure 6.6b	CPU Usage per Encryption/Decryption Operation at 4.9MB on Mac and Pi (Lower is Better)	55
Figure 6.7a	Energy Estimation at 1MB for Mac and Pi (Lower is Better)	56
Figure 6.7b	Energy Estimation at 4.9MB for Mac and Pi (Lower is Better)	56

## LIST OF TABLES

<b>Table Number</b>	<b>Title</b>	<b>Page</b>
Table 2.1	Literature Review of the Existing Methods	17
Table 2.2	Strengths and Weakness of the Project	18
Table 5.1	Specifications of Laptop	35
Table 5.2	Specifications of Raspberry Pi Zero 2 W	36
Table 5.3	Software Tools and Libraries Installed	37
Table 6.1	Metric Tracked and Description	46

## **LIST OF SYMBOLS**

-

-

## LIST OF ABBREVIATIONS

<i>AES</i>	Advanced Encryption Standard
<i>AES-128</i>	AES with 128-bit encryption key
<i>AES-256</i>	AES with 256-bit encryption key
<i>CC20</i>	ChaCha20 stream cipher
<i>Twofish</i>	A symmetric key block cipher
<i>API</i>	Application Programming Interface
<i>CPU</i>	Central Processing Unit
<i>GPIO</i>	General Purpose Input Output
<i>IOT</i>	Internet of Things
<i>JSON</i>	JavaScript Object Notation
<i>RAM</i>	Random Access Memory
<i>MQTT</i>	Message Queuing Telemetry Transport
<i>RPi/Pi</i>	Raspberry Pi
<i>PKCS7</i>	Public-Key Cryptography Standards #7 (padding scheme)
<i>I2C</i>	Inter-Integrated Circuit
<i>Mac</i>	Macintosh/Macbook
<i>UI</i>	User Interface
<i>GUI</i>	Graphical User Interface

## **CHAPTER 1**

### **Introduction**

The rise of Internet of Things (IoT) sensor networks in Malaysian agriculture has transformed traditional farming practices by enabling farmers to collect real-time environmental data. These networks gather environmental information like soil moisture levels and temperature to assist farmers in improving crop productivity and managing resources efficiently [1]. Despite its significance, safeguarding the security of this sensor data still poses a persistent challenge that is frequently underestimated.

Smart farming uses technology to modernise traditional agriculture practices by offering immediate data to improve decision-making processes. In Malaysia, the integration of these advancements has played a role in tackling farming issues, such as erratic weather conditions, insect invasions, and limited resources. For example, the use of sensors allows ongoing monitoring of soil quality, resulting in accurate watering techniques that save water and enhance crop productivity and health. Furthermore, temperature and humidity sensors play a role in maintaining the perfect environment for crops to thrive [2]. By ensuring conditions for growth, these sensors help minimise the risk of crop losses and boost productivity as a whole.

The advantages of utilising farming go beyond just improving efficiency and productivity in agricultural practices. In Malaysia, farmers can use data analysis to predict and address potential challenges before they escalate into major concerns. This proactive strategy not only boosts crop production but also reduces the environmental footprint of farming methods.

Despite these progressions in technology, there is a concern surrounding the protection of agricultural sensor information. The transmission of agricultural data without encryption, such as soil moisture levels, temperature recordings, and nutrient content, poses potential risks for farms. Additionally, unauthorised access to agricultural data can disrupt automated systems, such as irrigation controllers, resulting in significant operational challenges [3]. Addressing these security concerns is essential to safeguarding the integrity and reliability of smart agriculture systems in Malaysia.

## 1.1 Problem Statement and Motivation

In the changing world of smart farming, Malaysia's agricultural sector has embraced modern farming techniques that rely on data from IoT sensors on their farms to enhance their farming practices efficiently by gathering crucial information about the environment, like soil moisture content and temperature to help improve crop productivity and resource utilization effectively.

On the other hand, transmitting this important agricultural information is frequently done without encryption, which poses major security risks. Unprotected data pathways leave farms vulnerable to potential threats like data breaches, alteration of crop conditions, interruptions in automated processes and an overall compromise in farm operations. It is crucial to tackle this pressing security issue to safeguard the trustworthiness and dependability of farming practices, in Malaysia.

## 1.2 Objectives

1. To develop a secure and cost-effective data transmission system for smart agriculture using Raspberry Pi as the target platform, MQTT and Node-RED
2. To implement and benchmark multiple encryption algorithms (AES-128, AES-256, ChaCha20, and Twofish) for securing sensor data in transit.
3. To evaluate the performance of these algorithms in terms of encryption time, decryption time, memory usage, and CPU usage on resource-constrained hardware.

These objectives collectively aim to deliver a replicable and secure communication framework suitable for real-time monitoring in small to medium-sized smart farming environments.

## 1.3 Project Scope

The scope of this project involves the development and implementation of a secure IoT-based data collection system for environmental monitoring within agricultural contexts. Environmental sensor readings will be emulated using Python scripts on both the development machine and Raspberry Pi, and transmitted as encrypted payloads to a Node-RED flow via MQTT. The Node-RED flow will include decryption logic for multiple algorithms and present visual feedback through an interactive dashboard.

The project also includes a comparative evaluation of the selected encryption algorithms (AES-128, AES-256, ChaCha20, and Twofish), focusing on metrics such as processing time, memory usage, and CPU load. A key consideration is ensuring the system remains lightweight, modular, and suitable for future integration with actual sensors. This supports the broader aim of creating a practical and scalable solution for secure data transmission in small-scale agricultural IoT deployments.

The project does not include the use of actual physical sensors; instead, all environmental data is emulated through Python scripts. Additionally, cloud-based data storage, remote access capabilities, and advanced analytics are outside the scope of this work. The implementation also excludes security components such as key exchange protocols, public key infrastructure (PKI), and user authentication mechanisms. Network-level security threats, including denial-of-service (DoS), man-in-the-middle (MITM) attacks, and packet sniffing, are not addressed. Furthermore, energy consumption metrics are estimated using CPU usage and processing time, as external power measurement tools are not utilised.

### **1.4 Contributions**

This project addresses vulnerability concerns in protecting sensitive agricultural data within the agriculture sector in Malaysia. By integrating open-source tools like Raspberry Pi and Node-RED with robust encryption, the project significantly improves the security of sensitive agricultural data transmission, preventing interception, manipulation, and unauthorised access. In addition to demonstrating encryption workflows and decryption handling, it benchmarks the performance of various cryptographic methods, aiding future IoT developers in selecting suitable algorithms. This solution improves trust, reliability, and scalability of smart farming systems in resource-constrained environments.

### **1.5 Report Organization**

The thesis is organized into 7 Chapters briefly discussed below:

Chapter 1 is divided into 5 sections. The first section discusses the problem statement and motivation behind the research. Section 2 lists the research objectives. Section 3 lists the



## Chapter 1 Introduction

project scope and objectives. Section 4 discusses the contributions of the project to the field of smart agriculture. Finally, Section 5 provides an overview of the report organisation.

Chapter 2 is split into 4 sections. Section 1 provides a review of relevant technologies including Raspberry Pi, Node-RED, MQTT, Python, encryption algorithms, and monitoring tools. Section 2 discusses previous works related to secure smart farming. Section 3 highlights the limitations identified in existing studies. Section 4 offers a summary of the literature reviewed.

Chapter 3 is divided into 7 sections. Section 1 introduces the system design through various diagrams. Section 2 presents the tools and technologies used. Section 3 details the secure data transmission workflow. Section 4 explains the decryption and data visualisation process using Node-RED. Section 5 outlines the benchmarking methodology. Section 6 describes the data collection and analysis process. Section 7 concludes with a summary of the methodology.

Chapter 4 is split into 4 sections. Section 1 presents the overall system block diagram. Section 2 provides the specifications of the hardware and software components. Section 3 focuses on the design of circuits and emulated sensors. Section 4 describes how different system components interact in transmitting and receiving sensor data.

Chapter 5 is divided into 6 sections. Section 1 discusses the physical hardware setup. Section 2 explains the software setup and environment configuration. Section 3 elaborates on system configurations, particularly encryption and MQTT settings. Section 4 demonstrates system operation with illustrative screenshots. Section 5 identifies challenges encountered during implementation. Section 6 concludes the chapter with final remarks on the implementation phase.

Chapter 6 is separated into 5 sections. Section 1 outlines the testing objectives and performance metrics. Section 2 presents the testing setup and detailed results for each metric. Section 3 discusses project challenges related to performance and resources. Section 4 evaluates how well the objectives were met. Section 5 concludes the discussion on system evaluation.

Chapter 7 is divided into 2 sections. Section 1 summarises the main findings and conclusions drawn from the project. Section 2 provides recommendations for future improvements and potential research directions.

## Chapter 2

### Literature Review

#### 2.1 Review of Technologies

The section highlights various technologies both hardware and software that will be utilised for the proposed project.

In this chapter, the project will highlight the related literature review of the technology used.

##### 2.1.1 Raspberry Pi

The Raspberry Pi is a compact, cost-effective single-board computer widely utilised in IoT applications due to its versatility and energy efficiency. Its compatibility with various sensors and support for multiple programming languages make it ideal for smart agriculture systems. The Raspberry Pi's ability to handle data collection, processing, and transmission tasks efficiently aligns with the project's requirements for a secure and scalable agricultural monitoring solution.

##### 2.1.2 Node-RED

Node-RED is a flow-based development tool designed for visual programming, particularly suited for IoT applications. Developed by IBM, it allows for the seamless integration of hardware devices, APIs, and online services. In this project, Node-RED facilitates the creation of dashboards for real-time data visualisation and management, enhancing user interaction and system monitoring capabilities.

##### 2.1.3 MQTT Protocol

MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol optimised for high-latency or unreliable networks, making it ideal for IoT environments. It operates on a publish-subscribe model, ensuring efficient data transmission between devices. The protocol's minimal bandwidth requirements and low power consumption are advantageous for resource-constrained devices like those used in smart agriculture.

### 2.1.4 Python Programming Language

Python is a high-level, interpreted programming language known for its readability and extensive library support. Its versatility makes it suitable for various aspects of IoT development, including data processing, automation, and implementing encryption algorithms. In this project, Python is employed to develop scripts for data encryption and decryption, leveraging libraries such as PyCryptodome to ensure secure data handling.

### 2.1.5 Encryption Algorithms

Ensuring data security in IoT applications is paramount, especially in sectors like agriculture where sensitive information is transmitted. This project evaluates several encryption algorithms to determine their suitability for resource-constrained environments:

- **AES (Advanced Encryption Standard):** A symmetric encryption algorithm known for its robustness and efficiency. Variants like AES-128 and AES-256 offer different key lengths, balancing security and performance.
- **ChaCha20:** A stream cipher designed for high performance in software implementations, offering strong security with faster processing times compared to traditional algorithms.
- **Twofish:** A symmetric key block cipher recognized for its flexibility and speed, making it a viable option for devices with limited computational resources.

Studies have benchmarked these algorithms on platforms such as the Raspberry Pi, assessing their performance in terms of speed, memory usage, and energy consumption.

### 2.1.6 Data Visualisation and Monitoring Tools

Effective data visualisation is crucial for monitoring agricultural parameters and making informed decisions. Tools integrated within Node-RED enable the creation of interactive dashboards, providing real-time insights into environmental conditions. These dashboards facilitate the tracking of metrics such as soil moisture, temperature, and humidity, allowing for timely interventions and resource optimization.

## **2.2 Related Works in Secure Smart Farming**

The report will highlight the existing works and project how our proposed solution fills identified gaps and advances the field.

### **2.2.1 Precision Agriculture Monitoring System using Wireless Sensor Network and Raspberry Pi Local Server**

Work by the author Flores [4] underlines the need for real-time monitoring possible through WSN in reducing risks to production due to both environmental and human factors. Their study delimited how there is a need to take advantage of the field data and make it possible for farmers to make necessary adjustments in crop production timely in order to increase agricultural productivity and resilience.

With more demands for the increased production of food, there has been growing economic pressure on investors to adopt aggressive farming methods. While this brings high yields in the short run, it depletes the natural resources. The adoption of sustainable agricultural practices through the use of new technologies is required if responses to such challenges are to be affected. Poor utilization, due to a lack of information, awareness, and resistance to the adoption of new technologies outweighs the effectiveness of new technologies. Author Flores [4] expresses that in order for environmental monitoring to encourage sustainable agriculture, there is a dire need to develop low-cost, user-friendly, efficient monitoring systems.

Relevant to this paper is the Pods project at the University of Hawaii whose objective is the ecological environment and events monitoring around rare plants. This project would deploy attached micro weather sensors to communication units, or "pods," to monitor sunlight, temperature, wind, and rainfall. Considering a low cost and low interference with the terrain, the pods form a wireless ad-hoc sensor network capable of transmitting data by themselves and forwarding it to other nodes. Implementation of the wireless routing protocol MOR (Multi-path On-demand Routing Protocol) comes true with maximized efficiency in routing and energy conservation, enhancing scalability and robustness for the Sensor Network.

Among all the parameters that highly influence crop growth, temperature, humidity, soil moisture, and soil pH are considered to be some of the key parameters in agriculture monitoring sensors. Of these, soil electrical conductivity and pH bear high importance because of their critical role in characterizing field variability and optimizing precision

agriculture practices. According to the author Flores [4], accurate measurement of soil pH is very important because it defines nutrient availability and plant growth. This study further renders the importance of pH extremes concerning plant tolerance and nutrient uptake, emphasizing that the soil pH should be optimally adequate for agricultural productivity.

Testing sensors is an important aspect that leads to full assurance of the accuracy and reliability of agriculture sensors. Flores conducted in-depth testing of numerous sensors, such as the DHT22 sensor for temperature and humidity, ALS-PT19 for light, soil moisture sensor, and analogue pH meter. Each sensor was subjected to all sorts of different environmental conditions to check for its accuracy and response. Additionally, there is an EC meter that was developed after rigorous testing to relate the EC analogue values and electrical conductivity measurements. Moreover, the study has established the calibration and validation of sensors as key factors in guaranteeing agricultural data integrity.

Eventually, it has been observed from the reviewed literature how much more significant WSNs and state-of-the-art sensor technologies play regarding revolutionary changes in agricultural monitoring and management. Several such works have highlighted real-time data acquisition, sustainability of agriculture, and sensor reliability as critical aspects that determine improvements in agricultural productivity and resilience. By integrating novel technologies with sound sensor testing methodologies, scholars will be in a position to achieve affordable and effective solutions towards overcoming farming challenges and improving food production in a sustainable manner.

In light of this, an evaluation of the strengths and weaknesses of individual studies reveals both promising approaches and notable gaps. Flores et al. [4] present a compelling concept for agricultural monitoring. Its strengths lie in the use of wireless sensor networks (WSNs) combined with a Raspberry Pi local server, which enables live monitoring of environmental variables to facilitate timely agricultural decisions. The project emphasises low-cost implementation using commonly available hardware components, making it financially suitable for farmers and promoting the widespread adoption of precision agriculture practices.

However, several limitations can be observed. The project's reliability is heavily dependent on the durability and stability of its hardware components, such as the Raspberry Pi, sensors, and the central server. Sensor failures or connectivity issues could disrupt data collection and negatively impact data-driven decision-making. Additionally, while the system handles

sensitive agricultural data, the study [4] does not explore data protection mechanisms. As such, adequate measures to safeguard this data against unauthorized access or cyber-attacks would be necessary for real-world deployment.

### 2.2.2 Cyber Attack on Smart Farming Infrastructure

The author Sontowski [5] shows the threatening aspect brought about by cyber attacks on smart farming infrastructure; this paper describes the implementation of denial-of-service (DoS) attacks targeting a smart farm architecture that is connected to the 2.4 GHz network. Namely, a Wi-Fi deauthentication attack managed to totally disrupt the communication between the Raspberry Pi and the Wi-Fi access point and, by implication, prevent data from going to the Azure cloud. Moreover, using the MakerFocus ESP8266 Development Board WiFi Deauther Monster, the attackers expand the attack to disable any devices attempting to connect. Hence, disconnecting the Raspberry Pi from the network is achieved, as highlighted in Figure 2.1. This particular attack utilises, among others, deficiencies of the IEEE 802.11 protocols, such as the sent-in plaintext management frames, which render them susceptible to deauthentication frames containing spoofed MAC addresses that forge the AP. The study emphasises that such types of attacks are quite difficult to detect since they bypass the traditional systems of security such as MAC filtering and intrusion detection systems.

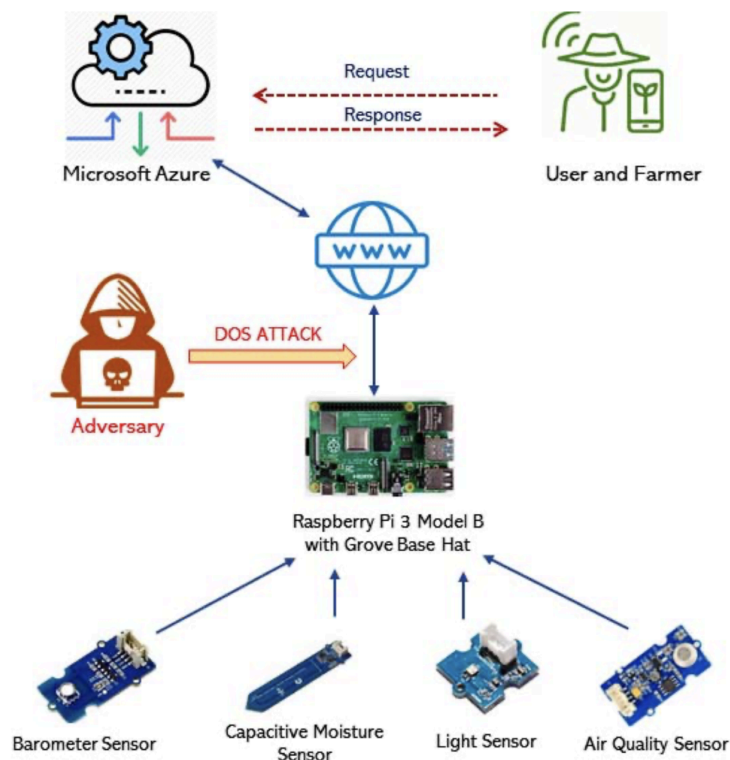


Figure 2.1: System Architecture and Attack Vector [5]

Figure 2.1 highlights a potential cybersecurity threat, an adversary launching a DDOS attack against the Raspberry Pi. This attack could disrupt the system's performance, blocking communication of data between the sensors, cloud platform, and end users, thereby compromising the confidentiality and integrity of smart agriculture.

A very critical example concerning the importance of understanding and mitigating such vulnerabilities in the infrastructure of smart farming is the Wi-Fi deauthentication attack execution by the author Sontowski [5]. It can be executed through tools that are easily available, targeting the weaknesses within the 802.11 protocol, posing an immense danger towards the reliability and security of the agricultural sensor network. The successful attack demonstrates the implementation of robust security measures, such as encryption of management frames by IEEE 802.11w, which prevents spoofing and unauthorized access through a method of encrypting management frames, as highlighted in Figure 2.2. Furthermore, this study underlines that the challenges of dealing with cybersecurity will not be confined only to smart farming but to each other domain of IoT, where similar vulnerabilities might exist.

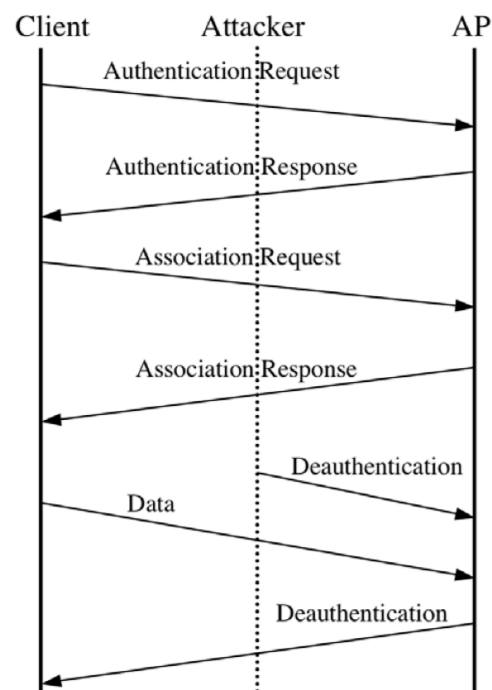


Figure 2.2: Graphical Depiction of a Deauthentication Attack [5]

Figure 2.2 gives a graphic representation of the deauthentication attack by relating the client, attacker, and access point in such a way that the attacker intercepts the regular flow of communication from authentication and association in an unencrypted management frame and subverts to send deauthentication frames that disrupt the established connection between client and access point, hence keeping the client disconnected from the network.

It has been established from the review of the literature that cybersecurity plays an imperative role in the infrastructure of smart farming, thus creating a platform for developing proactive measures against cyber-attacks. This benefits both researchers and practitioners by allowing them to understand the methodologies and implications of such malicious threats, develop appropriate security strategies, and deploy them effectively to protect agricultural systems.

Building on this perspective, an analysis of the strengths and weaknesses of relevant cybersecurity studies reveals critical vulnerabilities and defensive approaches. The study by Sontowski et al. [5] addresses a highly relevant aspect of modern smart farming infrastructure: cybersecurity. One of the key strengths of this work lies in its demonstration of a practical Wi-Fi deauthentication attack using the MakerFocus ESP8266 Development Board WiFi Deauther Monster. The authors [5] provide empirical evidence of vulnerabilities in 2.4 GHz networks, commonly used in smart agriculture, by exposing exploitable management frames in the IEEE 802.11 protocol. These findings highlight the potential for spoofed deauthentication frames to compromise network availability.

Despite its relevance, the study has certain limitations. It focuses exclusively on Wi-Fi deauthentication attacks within the 2.4 GHz frequency band. While this exposes one significant vulnerability, it does not consider other possible cyber threats to smart agriculture systems, such as man-in-the-middle attacks, data injection, or physical tampering. Moreover, although the study recommends implementing IEEE 802.11w as a countermeasure, it does not present a comprehensive set of mitigation strategies or technical implementations for broader threat coverage.

### **2.2.3 Securing the Internet of Battlefield Things with ChaCha20**

Navalino et al. [6] proposed the use of ChaCha20-Poly1305 encryption to enhance data security for resource-constrained devices operating within the Internet of Battlefield Things

Bachelor of Information Technology (Honours) Communications and Networking  
Faculty of Information and Communication Technology (Kampar Campus), UTAR



(IoBT) environment. The study focused on applying lightweight cryptography in real-time sensor communication using a Raspberry Pi Pico microcontroller and nRF24L01 radio modules. Although the original context is military-based, the underlying architecture and methodology are directly applicable to smart agriculture systems, where energy efficiency and secure data transmission are critical requirements.

The authors [6] evaluated encryption and decryption performance by measuring the execution time, throughput, and avalanche effect using live sensor data from MPU6050 modules. Encryption times ranged from 261 ms for 16-byte messages to 17,472 ms for 8,192-byte messages. Despite the increasing data size, the throughput remained relatively stable, peaking at around 468 Bps for encryption and 465 Bps for decryption, respectively. An average avalanche effect of 50.53% was recorded, indicating strong resistance to cryptanalytic attacks with only minor input changes.

The implementation of ChaCha20-Poly1305 demonstrated suitability for real-time environments due to its lightweight nature, high speed, and consistent output performance. This study is relevant to the current project as it provides empirical justification for using ChaCha20 as an alternative to AES in low-power, low-latency IoT applications such as smart agriculture. The findings [6] support the benchmarking approach in this project, particularly in comparing symmetric encryption algorithms under constrained hardware conditions.

In evaluating the contributions of this work, Navalino et al. [6] presents several notable strengths. Firstly, it evaluates the ChaCha20-Poly1305 encryption algorithm within a real-world embedded environment using the Raspberry Pi Pico, along with actual sensor data transmission. This approach aligns well with the goals of the current project, which involves performance benchmarking of lightweight encryption in constrained IoT systems. The study's analysis of encryption and decryption timings, throughput measurements, and avalanche effect provides comprehensive insight into the algorithm's practical viability. Additionally, the consistent throughput across varying payload sizes demonstrates the algorithm's efficiency for systems with variable data loads, such as those found in agriculture.

On the other hand, the study does have some limitations. It focuses solely on ChaCha20-Poly1305 and does not include performance comparisons against other algorithms like AES or Twofish, which are included in this project. This reduces its value as a comparative reference across multiple encryption schemes. Furthermore, the research is

positioned within a military IoT context (IoBT), which, while sharing technical similarities with agricultural IoT, may not fully reflect the environmental and data usage conditions of smart farming systems. Nonetheless, the hardware and security constraints discussed in [6] are largely equivalent, allowing the findings to be transferable and still relevant to this project's objectives.

### **2.2.4 Comparative Performance Analysis of Lightweight Cryptography Algorithms**

Fotovvat et al. [7] conducted a comprehensive study titled “Comparative Performance Analysis of Lightweight Cryptography Algorithms for IoT Sensor Nodes,” focusing on evaluating encryption performance in real-world IoT environments. The study compares 32 authenticated encryption with associated data (AEAD) algorithms, including AES-GCM, AES-CCM, and other lightweight cipher suites across three embedded platforms: Raspberry Pi 3B, Raspberry Pi Zero W, and the iMX233 board.

The researchers [7] measured execution time, RAM usage, and energy consumption for each algorithm using a standardized testing environment. They highlighted how AES-based modes like GCM and CCM, while offering high security, consume significantly more resources compared to newly developed lightweight algorithms. This is crucial in IoT systems where power efficiency, low latency, and memory constraints are fundamental design considerations.

Fotovvat et al. [7] also deployed the algorithms in a practical IoT sensor node scenario, encrypting 30-byte sensor payloads and transmitting them over LoRa communication. They found that encryption time typically accounted for only 5–10% of total transmission time, demonstrating that selecting an efficient cipher can meaningfully impact overall system energy usage and responsiveness.

While this study primarily focused on NIST's LWC candidates, it provides valuable insight into how classic algorithms like AES (and by extension, Twofish and ChaCha20) compare in performance under embedded constraints. This aligns closely with the objective of the present project, which benchmarks various encryption algorithms in a smart agriculture environment using Raspberry Pi Zero W and Node-RED for secure MQTT communication [7]. Fotovvat et al. [7] contribute several strengths that reinforce the relevance of their work to this project. Notably, the study includes extensive benchmarking of 32 AEAD algorithms across multiple embedded platforms, including the Raspberry Pi Zero W, which is also used

in this implementation. This direct hardware alignment ensures the performance insights are highly applicable. Secondly, the inclusion of a real-world IoT deployment scenario, where encrypted sensor data is transmitted over a LoRa interface, adds significant practical value. It illustrates how encryption affects total system performance in terms of timing and energy use, especially in constrained devices commonly found in agriculture IoT systems.

Moreover, the study provides quantitative analysis on core performance metrics—execution time, RAM usage, and power consumption—that mirror the benchmarking goals of this project. This makes the literature a strong foundation for supporting algorithm selection in resource-limited environments such as smart farms.

However, there are a few limitations to consider. While the study compares a broad range of lightweight cryptographic algorithms from the NIST LWC standardization process, it does not explicitly evaluate ChaCha20 or Twofish, two algorithms used in this project. As a result, while the benchmarking methodology is relevant, the direct applicability to those specific ciphers is somewhat limited. Additionally, the encryption tests primarily focus on small payloads (~30 bytes), which, although typical in sensor networks, may not fully represent systems that handle larger or variable-length data. Lastly, the study focuses exclusively on symmetric AEAD encryption, and does not explore hybrid or asymmetric encryption models, which may be relevant for some smart agriculture systems that involve cloud-based services or device-to-device authentication.

Despite these limitations, the paper remains a valuable and contextually appropriate reference for this project, especially in justifying the need for performance-aware encryption selection in IoT-based agriculture.

### 2.3 Limitations of Previous Study

Table 2.1 provides a summary of the existing literature and their identified limitations in relation to the secure smart agriculture system.

Table 2.1 Literature Review of the Existing Methods

Year	Author(s)	Technique	Problems	Limitations
2018	Flores et al.	Smart farming using IoT and Raspberry Pi	Real-time sensor monitoring and automation in agriculture	Does not implement security or encryption mechanisms

2021	Sontowski et al.	Analysis of cybersecurity vulnerabilities in smart agriculture	Identifies real-world attack vectors and risk factors	No solution or implementation details provided
2024	Navalino et al..	ChaCha20-Poly1305 encryption for secure IoT transmission	Ensures secure communication in restricted environments	Focuses only on CC20; lacks algorithm comparison
2021	Fotovvat et al.	Performance benchmarking of 32 lightweight AEAD encryption algorithms	Evaluates execution time, energy use, and memory on platforms	Does not include Twofish of CC20; no implementation case study

Table 2.1 outlines the various projects' techniques, problems, and limitations in the context of project themes, ranging from real-time monitoring and data security, encryption algorithm benchmark to IoT integration and vulnerabilities, highlighting their specific strengths and challenges. Table 2.2 illustrates the benefits and challenges of the project.

Table 2.2 Strengths and Weakness of the Project

Author(s)	Strengths	Weakness
2.1.2 Flores et al.	Provides a comprehensive real-world architecture for smart farming using IoT and Raspberry Pi. Demonstrates the feasibility of low-cost agricultural monitoring systems.	Does not address encryption or data security. Focuses primarily on system deployment and monitoring functions.
2.2.2 Sontowski et al.	Highlights real security threats in smart farming, including DoS and Wi-Fi interception. Validates the need for cryptographic security in agricultural IoT.	Lacks technical implementation of benchmarking of security protocols. Mostly theoretical.
2.3.2 Navalino et al..	Demonstrates practical use of ChaCha20-Poly1305 on Raspberry Pi Pico with real-time sensor data. Includes encryption/decryption time, throughput, and avalanche analysis.	Focuses solely on ChaCha20; no comparison with AES or other algorithms. Context is military (IoBT) rather than agricultural IoT
2.4.2 Fotovvat et al.	Offers large-scale benchmarking of 32 lightweight encryption algorithms on Raspberry Pi Zero W. Includes metrics like energy	Does not include ChaCha20 or Twofish. Primarily focus is AEAD algorithms under NIST LWC without

	use, RAM consumption, and execution time.	application-level integration
--	---	-------------------------------

Table 2.2 exposes the strengths and weaknesses of various projects focused on integrating technology in agriculture, specifically in the contexts of environmental monitoring, cybersecurity, encryption benchmark, and IoT applications.

### 2.4 Summary

In summary, this chapter has reviewed a range of literature relevant to the implementation of secure IoT systems in agriculture. The studies examined offer insights into IoT agriculture, cybersecurity concerns, and lightweight encryption algorithms applicable in this environment. Early works by Flores et al. [4] highlighted the feasibility of IoT integration in agricultural monitoring, while Sontowski et al. [5] addressed emerging cybersecurity risks associated with smart farming systems. Navalino et al. [6] and Fotovvat et al. [7] contributed valuable findings on encryption performance in embedded platforms, offering practical benchmarking data and considerations for selecting appropriate cryptographic techniques. These reviewed studies form the foundation for the selection of encryption techniques in the proposed secure sensor data transmission system.

## Chapter 3

### System Methodology

In this chapter, the method that gives the solution of the entire project is identified.

#### 3.1 System Design Diagram

The system is designed to securely transmit environmental sensor data using various encryption algorithms, simulating a smart agriculture use case. The synthetic data used in this project to simulate environmental readings is generated through a Python script, encrypted based on the selected algorithm, and transmitted via the MQTT protocol. Upon reception, Node-RED handles decryption and visualisation of the sensor metrics. This section presents the high-level system design through architecture, use case, and activity diagrams to illustrate the flow and interactions within the system, starting with Figure 3.1 System Architecture Diagram.

##### 3.1.1 System Architecture Diagram

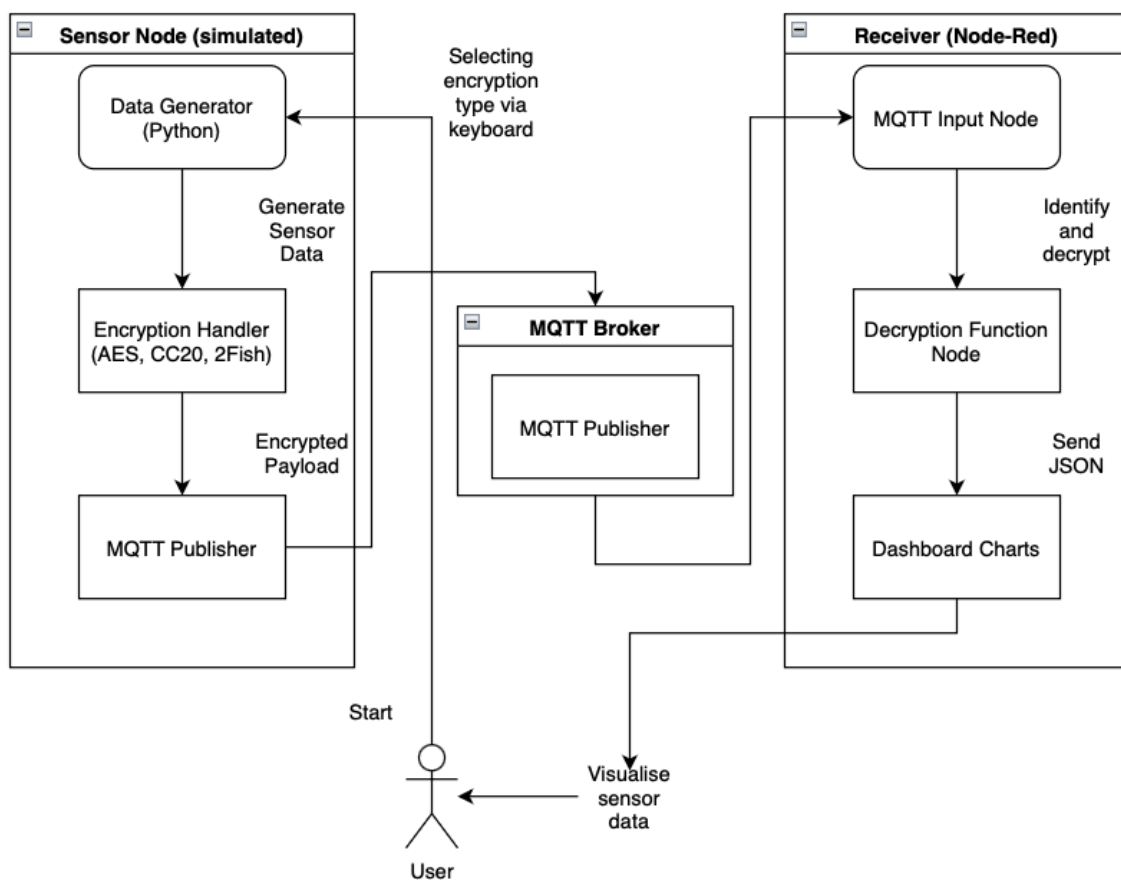


Figure 3.1: System Architecture Diagram

Figure 3.1 shows the overall structure of the architecture from sensor simulation and encryption to MQTT messaging and dashboard visualisation.

### 3.1.2 Use Case Diagram and Description

This diagram in Figure 3.2 outlines the interactions between the users, system, and components.

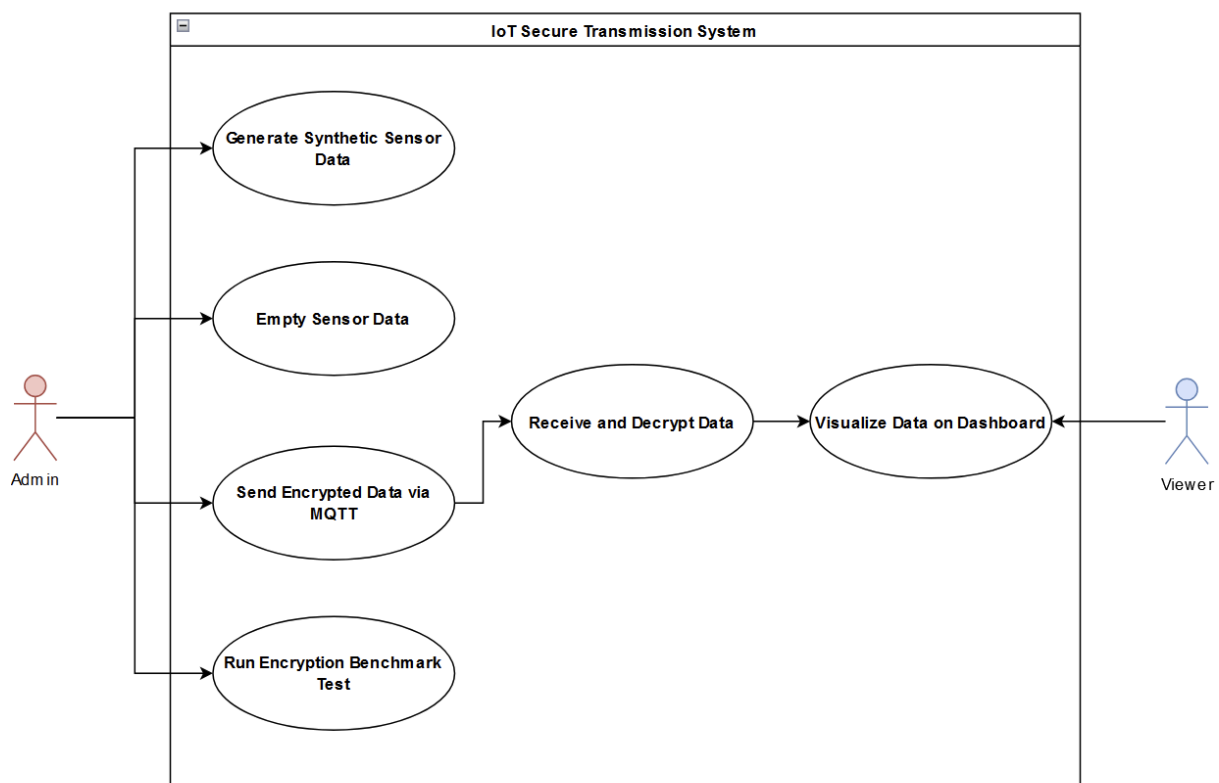


Figure 3.2: Use Case Diagram

In Figure 3.2 illustrates the roles of two user types, System Admin and Viewer and their interactions with key system functionalities. The System Admin initiates core processes, including generating synthetic sensor data, encrypting it with a selected algorithm, transmitting it securely via MQTT, and running encryption benchmark tests. Once the encrypted data reaches Node-RED, the system decrypts the payload and prepares it for presentation. The Viewer accesses the visualised data through a real-time dashboard, enabling observation of secure sensor information without direct interaction with encryption or transmission components.

### 3.1.3 Activity Diagram

Figure 3.3 will detail the step-by-step flow during data transmission and reception.

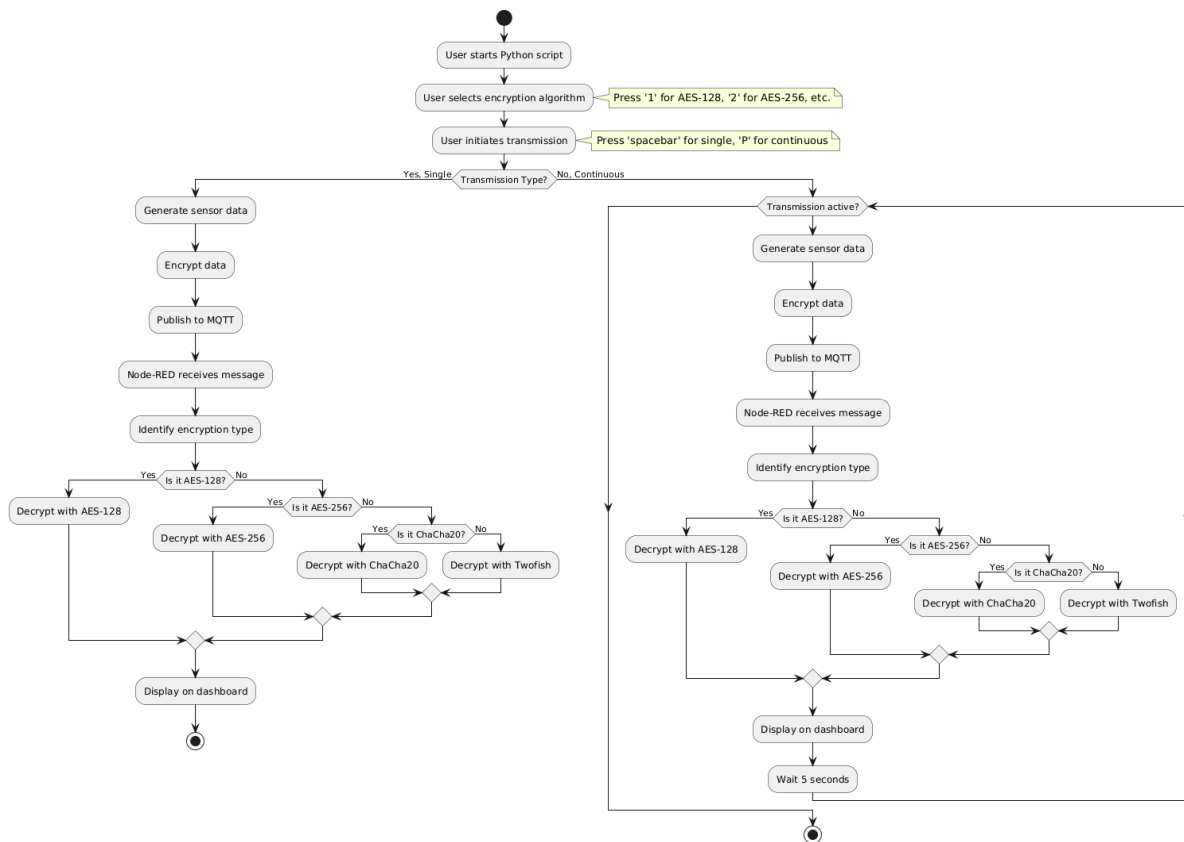


Figure 3.3: Activity Diagram

This activity diagram in Figure 3.3 illustrates the sequence of operations from the user selecting an encryption algorithm to the final data display on the dashboard. The process includes key decision nodes for algorithm selection, encryption type recognition, and decryption success. Once data is successfully decrypted by Node-RED, it is visualised live. Optionally, the user may execute a separate benchmarking script, shown independently from the main process flow to evaluate algorithm performance metrics without impacting the core transmission pipeline.



### 3.2 Tools and Technologies Used

This section outlines the hardware, software, and libraries used to develop and implement the secure data collection system. The technologies were selected based on suitability on IoT environments, encryption capabilities, and support for real-time data transmission and visualisation.

#### 3.2.1 Hardware

- **Raspberry Pi Zero 2 W**

Intended as the target platform for deployment, the Raspberry Pi offers a low-power, cost-effective solution for real-time data processing and sensor integration. It is capable of running Python scripts, handling encryption tasks, and communicating with MQTT brokers, making it ideal for embedded IoT environments such as smart agriculture [17].

- **MacBook (Simulated Environment)**

While the Raspberry Pi Zero 2 W is the target deployment platform, the MacBook served as a development environment during testing and a contributor in benchmarking stages. It was fully capable of running all system components, including encryption, MQTT publishing, and Node-RED flows. During this stage, it also served as a reliable environment for benchmarking encryption algorithms and performing visualisation via the local dashboard. Both the Raspberry Pi and MacBook are considered viable hardware for executing the system, depending on deployment requirements.

#### 3.2.2 Software Tools

- **Python 3.13**

Used as the primary programming language for scripting sensor data generation, encryption logic, and MQTT publishing. Its simplicity and compatibility with cryptographic libraries make it suitable for rapid development in IoT applications.

- **Node-RED**

A low-code, flow-based development tool used for MQTT data intake, decryption, and dashboard visualisation [14]. The function nodes in Node-RED were configured to dynamically detect the encryption type and apply the corresponding decryption

method. It also provides an integrated dashboard for monitoring the sensor data in real time.

- **Node.js and npm**

Required by Node-RED to support JavaScript libraries used in decryption flows. These packages ensure Node-RED is able to load required cryptographic modules via global context settings.

- **Mosquitto (MQTT Broker)**

A lightweight MQTT broker used for transmitting encrypted payloads from the Python script to Node-RED. It operates over the local network on the same machine during testing, simulating edge-device communication.

### 3.2.3 Python Libraries and Packages

- **paho-mqtt**

A Python client library used to implement MQTT publishing functionality. It allows the encryption script to push data to a broker on a specified topic.

- **PyCryptodome**

A self-contained Python package of low-level cryptographic primitives, used to implement AES-128 and AES-256 encryption in ECB mode. It supports padding schemes and byte-level operations required for secure encryption.

- **PyNaCl**

Python bindings to the Networking and Cryptography (NaCl) library. This was used for implementing ChaCha20 encryption via the 'SecretBox' method, offering secure, authenticated encryption for lightweight systems.

- **twofish**

A Python implementation of the Twofish cipher. It was used to evaluate a less commonly deployed encryption method for comparison with standard algorithms.

- **keyboard & threading**

The 'keyboard' library was used for capturing user input to switch between encryption modes, and 'threading' enabled the implementation of a continuous packet-sending loop.

- **matplotlib, psutil, tabulate**

These libraries were used in the benchmarking script to measure and visualise

encryption performance across algorithms. Metrics include CPU usage, memory consumption, and time-based throughput.

### 3.3 Secure Data Transmission Workflow

This section outlines how the system encrypts sensor data and transmits it securely through MQTT. The workflow begins with user input to select the encryption algorithm, followed by data generation, encryption, and structured publishing to the MQTT broker. Each payload includes both the encrypted data and metadata to ensure proper decryption on the receiver side.

#### 3.3.1 Encryption Mode Selection

The user initiates the process by selecting an encryption type using predefined keyboard keys. 4 modes are available:

- AES-128 (**key: 1**)
- AES-256 (**key: 2**)
- ChaCha20 (**key: 3**)
- Twofish (**key: 4**)

These selections determine which algorithm is applied for the next data packet sent. Pressing the 'spacebar' will send one packet. Pressing 'P' will keep sending packets every 5 seconds until pressing 'S' to stop.

#### 3.3.2 Sensor Data Generation

Fake environmental data is generated in JSON format to simulate agricultural parameters such as:

- Temperature
- Humidity
- Soil moisture
- Light intensity
- CO<sub>2</sub> levels

Each reading is randomized within realistic agricultural ranges.

### 3.3.3 MQTT Publishing

Each message is structured in JSON format with the following fields:

```
{  
  "encryptionType": "AES-128",  
  "encryptedData": "<Base64 encoded ciphertext>"  
}
```

The message is then published to the topic 'sensor/data' using the Paho MQTT client. This standard format ensures consistent decryption and identification on the receiver side.

### 3.3.4 Encryption Logic

Using conditional logic, the function node supports dynamic decryption based on the 'encryptionType' string in each message. This allows the system to process messages encrypted by any of the supported algorithms without modifying the node configuration.

The selected algorithm is applied to the generated data:

- **AES (128/256-bit):** Uses ECB [8] mode via PyCryptodome. Data is padded using PKCS#7 before encryption to mitigate diffusion properties and repetitive patterns.
- **ChaCha20:** Uses NaCl's 'SecretBox' with a 24-byte nonce [9]. Encrypted data is authenticated and includes both the nonce and ciphertext. The nonce ensures each message is unique, preventing replay attacks.
- **Twofish:** Encrypts padded data with PKCS#7 in 16-byte blocks using the Python twofish module [10]. SHA-256 is used to derive a compatible key size.

## 3.4 Node-RED Decryption and Visualisation Workflow

This section explains how incoming encrypted MQTT messages are processed within Node-RED. The workflow includes identifying the encryption type, decrypting the payload, parsing the data, and visualising the results using a real-time dashboard.

### 3.4.1 MQTT Message Reception

Encrypted payloads are received in Node-RED via an MQTT input node subscribed to the topic 'sensor/data'. Each message is expected to contain a JSON object with:

- **encryptionType:** Identifies the encryption algorithm applied to the payload
- **encryptedData:** the Base64-encoded ciphertext

### 3.4.2 Decryption Flow

A function node is used to handle decryption. The node performs the following steps:

1. **Parse the payload:** Extract 'encryptionType' and 'encryptedData'
2. **Decode:** Base64-decode the encrypted string
3. **Select decryption method:**
  - AES-128/AES-256: Uses CryptoJS with ECB mode and PKCS#7 padding
  - ChaCha20: Uses 'sodium-native' from the Node-RED global context
  - Twofish: Uses the 'twofish-ts' package (a TypeScript implementation compatible with Node.js) injected into Node-RED via 'functionGlobalContext'. This allows the Function node to decrypt Twofish encrypted payloads received over MQTT.
4. **Output decrypted JSON:** The resulting object is passed to subsequent nodes

Decryption errors (e.g., incorrect padding, mismatched keys) are caught and logged through the node's warning system.

### 3.4.3 Dashboard Integration

Decrypted sensor readings are passed to individual chart and gauge nodes for real-time visualisation. These metrics include:

- Temperature (°C)
- Humidity (%)
- Soil Moisture (%)
- Light Intensity (lux)
- CO<sub>2</sub> Level (ppm)

Each metric is updated on the dashboard every time a new message is decrypted successfully.

## 3.5 Benchmarking and Methodology

This section explains the methodology used to evaluate the performance of each encryption algorithm in terms of speed, throughput, memory usage, CPU load, and energy estimation [7,12]. The goal is to determine the most suitable cipher for resource-constrained IoT environments.

### 3.5.1 Benchmarking Script Structure

A standalone Python script was developed to benchmark 4 encryption algorithms:

1. AES-128
2. AES-256
3. ChaCha20
4. Twofish

The script measures both encryption and decryption performance across multiple data sizes (1 KB up to 10 MB) and entropy conditions. Each result is recorded for further analysis.

### 3.5.2 Performance Metrics

For each test, the following parameters were collected:

1. **Execution Time (ms)**: Duration of encryption and decryption.
2. **Throughput (MB/s)**: Rate of data processed over time.
3. **Memory Usage (KB)**: RAM consumption before and after each operation.
4. **CPU Load (%)**: Monitored continuously during execution.
5. **Estimated Energy Usage**: Derived from CPU usage over time, as an indirect indicator of power efficiency.

As actual power draw was not measured with hardware instruments, CPU-based estimation serves as an indicative proxy for energy consumption

### 3.5.3 Testing Parameters

- **Data Sizes**: 1 KB, 10 KB, 100 KB, 1 MB, 4.9 MB (Optional: 5 MB and 10 MB for extended testing but may crash on Raspberry Pi)
- **Entropy Levels**:
  - High entropy: Random binary data (012345)
  - Low entropy: Repetitive or low-complexity data (11111)
- **Iterations**: Three repetitions per test case for statistical reliability and reduce random variation. The test results will be averaged and be examined this way.

### 3.5.4 Data Collection Tools

The benchmarking script uses the following Python libraries:

- ‘psutil’: To monitor CPU usage and memory footprint
- ‘time’: For precise execution timing
- ‘numpy, statistics’: For calculating averages and standard deviations
- ‘tabulate, json’: To export readable reports
- ‘matplotlib’: For plotting performance comparisons

### 3.5.5 Algorithm Executions

Each encryption method follows the same process:

1. Generate random data of defined size and entropy
2. Encrypt and decrypt the data using the selected algorithm
3. Record time, memory, and CPU usage
4. Validate decryption correctness (assert decrypted == original)
5. Repeat for multiple data sizes

### 3.5.6 Summary Results Output

The script exports results as:

1. Tabulated Summaries
2. JSON files (for storage or external processing)
3. PNG file of the graph
4. MatPlot GUI for interactive viewing of the result

## 3.6 Data Collection and Analysis Process

This section describes how performance data from the encryption benchmarking was recorded, structured, and prepared for evaluation. The process ensures consistent comparison across algorithms and supports the project’s aim to identify lightweight, secure encryption for smart agriculture IoT systems.

### 3.6.1 Structured Metric Recording

Each benchmark test logs the following for each algorithm, data size, and entropy condition:

- Encryption Time
- Decryption Time

- Memory Usage
- CPU Usage
- Data Throughput (MB/s)
- Energy Consumption Estimation

The script internally verifies the decryption result to ensure correctness and flags any errors using `'assert decrypted == original_data'`.

### 3.6.2 Tabulated Output

Using the `'tabulate'` library, test results are formatted into readable tables with:

- Algorithm Name
- Average Timing (Encrypt / Decrypt)
- Standard Deviation
- Throughput
- CPU and Energy Estimations

These summaries are printed to console and optionally saved as `'json'` logs for documentation.

### 3.6.3 Visualisation and Export

Results are export into two formats:

- `'json'`: For structured storage and reproducibility
- `'png'` (charts): Graphs showing time, throughput, and CPU performance using `'matplotlib'`

All files are time-stamped and named by encryption algorithm and entropy type.

### 3.6.4 Reproducibility

The script includes argument flags to:

- Select which algorithms to test
- Specify data sizes or entropy level
- Add larger data sets optionally
- Export or suppress visual output

This allows repeatable testing under controlled conditions, enabling direct comparisons.



### **3.7 Summary**

This chapter described the full development methodology of the proposed project, from system architecture to implement the encryption workflow and perform benchmarking. Multiple encryption algorithms were integrated into a modular transmission system and a custom Python benchmarking tool was developed to measure their effectiveness under various conditions. The methods described form the basis for the system's evaluation in later chapters, where the results will be assessed in detail.

## Chapter 4

### System Design

This chapter details the system design and diagrams for ease of understanding.

#### 4.1 System Block Diagram

The system block diagram shown in Figure 4.1 provides a high-level overview of the secure data transmission process implemented in this project. It outlines the sequence of interactions from data generation and encryption to transmission and visualisation using MQTT and Node-RED.

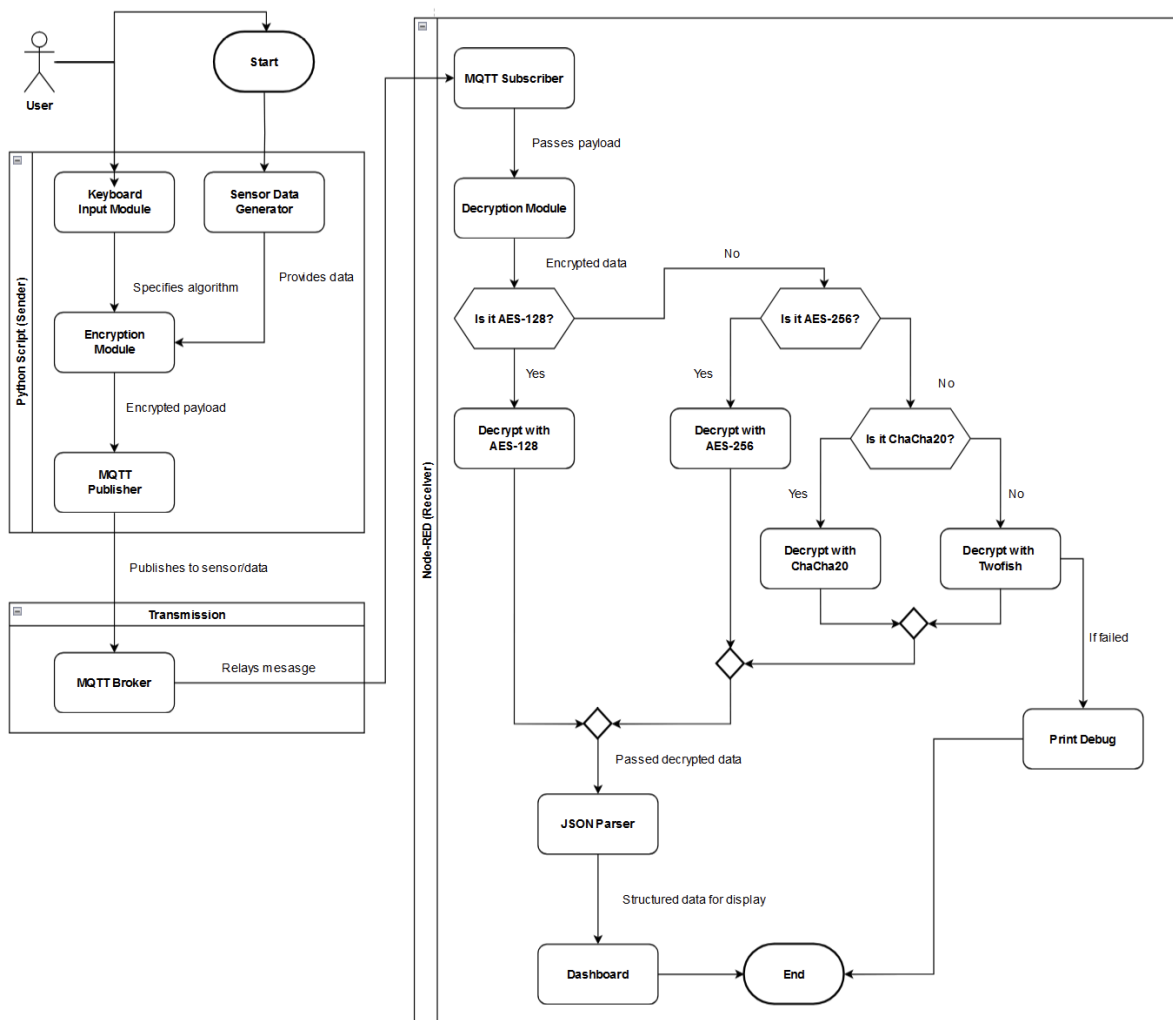


Figure 4.1: System Block Diagram

Figure 4.1 presents the complete system workflow, beginning with the user selecting an encryption method. A Python script then generates a random sensor value, which is encrypted and published via MQTT. The encrypted payload is transmitted through a broker and

received by Node-RED, which identifies the encryption type (determining if the encryption type is AES-128, AES-256, ChaCha20 or Twofish), decrypts the data, and displays the sensor readings on a dashboard.

### 4.2 System Components Specifications

This section outlines the essential hardware and software components used to implement the secure data transmission project. The chosen components are selected for their compatibility with IoT environments, lightweight operation, and support for real-time encryption and decryption message protocols.

#### 4.2.1 Hardware Components

##### 1. Raspberry Pi Zero 2 W

The Pi serves as the target hardware for deployment. It is compact, cost-effective, and capable of running the full encryption workflow including sensor simulation, algorithm switching, and MQTT publishing. It supports Python and can operate as an edge device for agriculture use.

##### 2. MacBook (Development and Testing Platform)

Throughout the development stage, a MacBook was used to simulate the full system. It served as the primary environment to test the encryption script, run the benchmarking tool, and host Node-RED for receiving and visualising sensor data. The performance of Apple Silicon allowed for reliable testing of all encryption algorithms before transitioning to the Pi.

#### 4.2.2 Software Components

##### 1. Python 3.13

Used for writing the main encryption script, handling sensor data generation, encryption mode selection, and MQTT publishing. It is also used to develop the benchmarking script that evaluates the encryption algorithm's performance.

##### 2. Node-RED (Flow-based Development Tool)

Node-RED was used to create the decryption and dashboard workflow. The tool processes incoming MQTT messages, dynamically detects the encryption type, and uses custom Function nodes to decrypt the payloads. Each sensor value is routed to real-time charts and gauge widgets on the dashboard.

### 3. Mosquitto (MQTT Broker)

A lightweight MQTT broker configured locally to simulate communication between the publisher (Python script) and subscriber (Node-RED). It enables encrypted data transmission using the ‘sensor/data’ topic

#### 4.2.3 Setup for Node-RED Function Global Context

To enable dynamic selection and execution of multiple decryption algorithms within Node-RED, certain cryptographic modules must be made globally accessible [14]. This is achieved through the `functionGlobalContext` configuration within the Node-RED `settings.js` file. By declaring the required modules in this context, the Function nodes within Node-RED can access them without needing to require the modules locally—something that is not supported natively within the Node-RED runtime for security and sandboxing reasons.

The following code snippet was added to the `functionGlobalContext` section of the `settings.js` file, located in ‘`cd ~/.node-red`’ on the terminal window :

```
“  
functionGlobalContext: {  
  CryptoJS: require("crypto-js"),  
  sodium: require("sodium-native"),  
  crypto: require("crypto"),  
  twofish: require("twofish-ts")  
}  
“
```

This setup allows the Node-RED Function nodes to access CryptoJS, sodium-native, the built-in Node.js crypto module, and twofish-ts, enabling them to decrypt messages based on the algorithm specified in the incoming payload.

However, this configuration step introduces a platform-specific challenge on macOS. In macOS, Node-RED instances installed via global npm or brew often run with **restricted access** to Node.js modules due to system integrity protections and permissions. As a result, modifying `settings.js` to load external libraries can lead to runtime errors or “module not found” issues if the modules are not properly installed in the correct context or if Node-RED is not executed with elevated permissions.

### 4.3 Circuits and Components Design

This project was primarily designed and tested in a simulated environment to validate the secure data transmission workflow. While no physical circuitry was implemented during the testing phase — aside from connecting the Raspberry Pi directly to the MacBook — the system remains fully compatible with standard agricultural sensors and is structured for future hardware integration.

#### 4.3.1 Simulated Sensor Emulation

Python script were used to emulate the output of real-world sensors by generating randomised values that mimic the behaviour of the following physical sensors:

- **DHT22:** For temperature and humidity readings
- **Soil Moisture Sensor:** For volumetric water content
- **Light Sensor:** (e.g. BH1750 or similar)
- **CO<sub>2</sub> Sensor:** To simulate environmental gas levels in ppm

These emulated values allowed for encryption benchmarking and visualisation in a realistic data flow without requiring actual physical sensors.

#### 4.3.2 Potential Hardware Integration

For future iterations or real-world development, the system can be easily adapted to read from physical sensors connected to a Raspberry Pi's GPIO pins. The Python script can interface with these devices using common libraries such as 'Adafruit\_DHT', 'smbus' for I2C, or serial communication.

This simulation-first approach enabled rapid development, consistent testing, and performance evaluation, while retaining hardware intercompatibility for smart agriculture applications.

#### 4.4 System Components Interaction Operations

This section describes how the various system components interact with each other during normal operation, from data generation and encryption to transmission, decryption, and proper visualisation.

##### 4.4.1 Sender Operations (Python Script)

The sender module begins execution upon user input, where the encryption mode is chosen using a keyboard. The script performs the following steps:

1. **Sensor Data Generation:** Synthetic environmental sensor data is created using randomized values within typical agricultural ranges.
2. **Encryption:** Based on user input (AES-128, AES-256, ChaCha20, or Twofish), the data is encrypted using the corresponding encryption library.
3. **Packet Structuring:** Encrypted data is encoded in Base64 and packaged into a JSON structure:

```
{  
  "encryptionType": "AES-256",  
  "encryptedData": "base64payload..."  
}
```
4. **MQTT Publishing:** The packet is published to the topic 'sensor/data' using Paho MQTT client on the Python script via the MQTT Broker with port 1883..

This modular and looped design enables continuous data output or on-demand packet dispatch, depending on user control.

##### 4.4.2 Message Transmission (MQTT Broker)

The Mosquitto broker acts as the intermediary, ensuring that all published MQTT messages from the sender are relayed to the Node-RED subscriber operating on a publish-subscribe model [15]. As both systems operate on the same local network, the broker guarantees low-latency transmission and accurate topic handling.

##### 4.4.3 Receiver Operations (Node-RED Flow)

On the receiving side, Node-RED listens for incoming messages on the topic 'sensor/data'. The data flow within the Node-RED is as follows:

1. **MQTT Input Node:** Receives the encrypted message and passes it downstream.

**2. Decryption Function Node:**

- Extracts the 'encryptedType' value
- Determine the correct type of encryption to pass the function
- Decodes the 'encryptedData'
- Applies the appropriate decryption function using 'CryptoJS', 'sodium-native', or 'twofish-ts' from the global context.

**3. Output:** Once decrypted, the JSON payload is parsed and routed to various chart and gauge nodes for display.

This interaction structure allows dynamic decryption and ensures a seamless transition from raw MQTT payloads to user-friendly dashboard elements.

**4.4.4 Dashboard Visualisation**

Each sensor metric (e.g., temperature, humidity, light intensity) is displayed on real-time charts or gauges within the Node-RED dashboard. These values are updated every time a new payload is successfully decrypted and parsed.

The flow is fully modular – any new encryption algorithm or data type can be integrated with minimal notifications to the Node-RED function node or dashboard layout.

## Chapter 5

### System Implementation

In this chapter, the implementation of the entire project is described.

#### 5.1 Hardware Setup

This section outlines the physical hardware setup used for implementing and testing the secure agricultural data transmission system. The setup was carried out in a simulated development environment, followed by hardware compatibility testing on Raspberry Pi.

Table 5.1 Specifications of Laptop

Description	Specifications
Model	Apple MacBook Air M1 A2237
Processor	M1 8-core CPU
Operating System	macOS 15 Sequoia
Graphic	M1 7-core GPU
Memory	8GB Unified Memory
Storage	256 GB SSD

Table 5.1 details the specification of the MacBook Air M1, primarily used as the development and simulation platform. All initial scripts including data generation, encryption, benchmarking, and Node-RED dashboard were developed and validated on the MacBook prior to hardware deployment. Its reliable performance allowed for seamless testing of multiple encryption algorithms.



Table 5.2 Specifications of Raspberry Pi Zero 2 W

Description	Specifications
Model	Raspberry Pi Zero 2 W
Processor	Quad-core 64-bit Arm Cortex-A53 @ 1 Ghz
Operating System	Raspberry Pi OS (Debian-based)
Graphic	VideoCore IV (64 MB LPDDR2 shared)
Memory	512MB LPDDR2
Storage	32 GB microSD Card

Based on Table 5.2, it details the hardware specifications of the Raspberry Pi Zero 2 W which was used as the target deployment device. The compact and power-efficient nature of the Raspberry Pi is suitable for use in IoT applications, specifically for real-time sensor processing, lightweight encryption, and MQTT-based data publishing and reception.

The combination of MacBook for development and Raspberry Pi for development testing ensures that the system is both prototyped efficiently and tested under realistic hardware constraints typical in agricultural environments.

## 5.2 Software Setup

The table below (Table 5.3) shows the software tools and libraries installed on both the development machine and Raspberry Pi Zero 2 W to support encryption, data handling, and visualisation. The mentioned software ensures seamless data generation, encryption, MQTT communication, and dashboard rendering in real-time with consideration of compatibility with lightweight systems and other cryptographic operations in IoT environments.

Table 5.3 Software Tools and Libraries Installed

Software/Package	Version	Purpose
Python	3.13	Main programming language for scripting and benchmarking
Node.js	22.x LTS	JavaScript runtime required by Node-RED and custom decryption modules
npm	10.9.2	Node.js package manager used to installed required libraries
Node.-RED	4.0.9	Flow-based tool for MQTT processing and dashboard rendering
Mosquitto MQTT Broker	2.0.21	Lightweight broker for MQTT data routing
PyCryptodome	3.19+	AES encrypt/decrypt using ECB mode and PKCS#7 pad
PyNaCl	1.5.0+	Secure CC20 implementation with SecretBox
Twofish (Python)	0.3.0	Python implementation of Twofish cipher
paho-mqtt	2.1.0	MQTT publishing client for Python
keyboard/threading	0.13.5/built-in	Handle user input and multithreaded packet looping
matplotlib, psutil, tabulate	Latest	Benchmark visualisation and system resource monitoring
crypto-js, sodium-native, twofish-ts	(npm modules) Latest	Decryption modules made globally available in Node-RED for Function Nodes

Table 5.3 describes the software and the libraries used for this project. It highlights the names of the package, the recommended version to install to operate correctly, and justification of their purpose in the project. Packages under npm will always be installed in the latest version due to its package manager behaviour.

### 5.3 Setting and Configuration

This section outlines the technical configuration steps required to prepare the development and testing environment for secure data transmission between devices. The setup includes configuring the encryption script, MQTT broker, Node-RED runtime, and dashboard flow necessary for real-time IoT communication. Each component was configured to simulate a typical smart agriculture deployment, with an emphasis on replicability and low system overhead.

#### 5.3.1 Encryption Script Configuration (Python)

To enable rapid switching and testing of encryption algorithms during benchmarking, the encryption script (sendAll.py) was modified to support in-script configuration via keyboard inputs. This minimized external dependencies and made the benchmarking process more efficient and controlled.

- **File:** sendAll.py
- **Encryption Keys:** hardcoded AES (128, 256) key value, ChaCha20 and Twofish also uses the same key but derived via SHA-256 hash
- **Keyboard Mappings:**
  - 1 = AES-128
  - 2 = AES-256
  - 3 = ChaCha20
  - 4 = Twofish
  - 'spacebar' = Send one packet
  - 'P' / 'S' = Start/Stop loop

No external configuration files were required, as all settings were handled within the Python script.

#### 5.3.2 Mosquitto MQTT Broker Setup

The MQTT broker facilitates lightweight message transmission over the local network using the publish-subscribe model. A local Mosquitto broker instance was used to emulate real-time data transmission between the sensor device and the dashboard.

- **Broker:** installed locally via 'apt install mosquitto' (Raspberry Pi) or 'brew install mosquitto' (MacOS)
- **Port:** 1883 (default)

- **Configuration File:** Not modified, default configuration sufficient for LAN
- **Broker Testing:** use ‘netstat -an | grep 1883’ to confirm broker operation

### 5.3.3 Node-RED Environment Configuration

- Installed via ‘npm install -g –unsafe-perm node-red’
- Accessed by visiting ‘<http://localhost:1880>’
- **Additional Node Packages Installed:**
  - node-red-dashboard
  - node-red-contrib-crypto-js
- Global context modules added to ‘settings.js’
  - functionGlobalContext: {  
CryptoJS: require("crypto-js"),  
sodium: require("sodium-native"),  
crypto: require("crypto"),  
twofish: require("twofish-ts")  
}

When configuring ‘settings.js’, ensure all required npm packages are installed with ‘npm install crypto-js sodium-native twofish-ts’.

After modification, Node-RED was restarted via: ‘node-red-stop && node-red-start’.

### 5.3.4 Flow Import and Dashboard Setup

To streamline testing and visualisation, a pre-configured JSON flow was imported into the Node-RED GUI. This flow handled incoming MQTT messages, decrypted them based on the selected encryption scheme, and displayed the results using visual dashboard widgets.

- Flow JSON file imported through Node-RED GUI
- Main node functions:
  - MQTT-in topic configured with ‘sensor/data’
  - Function Nodes: Match encryption type and decrypt
  - Chart/Gauge Nodes: Present decrypted readings in dashboard tabs

## 5.4 System Operation (with Screenshot)

This section outlines the runtime behaviour of the secure data transmission system. It describes the flow of data from the user-triggered encryption script to MQTT transmission,

decryption by Node-RED, and visualisation on a live dashboard. The operational flow showcases both the sensor simulation logic and the real-time performance of the message broker and dashboard interface.

### 5.4.1 Starting the Sender Script

The system begins when the user launches the Python script ('sendAll.py') on either the development machine or Raspberry Pi using 'python3 sendAll.py'. The user is presented with options to select an encryption:

- Press 1: AES-128
- Press 2: AES-256
- Press 3: ChaCha20
- Press 4: Twofish

Once selected, pressing 'spacebar' sends a single encrypted packet, while pressing 'P' will have the packet sent every 5 seconds until stopped by pressing 'S'.

Figure 5.1 will show a demonstration of the Python script running.

```
/usr/bin/python3 /Users/jiahuey/PycharmProjects/sendMac/sendAll.py
Press 1 for AES-128, 2 for AES-256, 3 for ChaCha20, 4 for Twofish.
Press SPACEBAR to send a single packet.
Press P to start continuous sending, S to stop.
Spacebar pressed. Sending a single packet.
```

Figure 5.1: Demonstration of Python Script

Figure 5.1 demonstrates the interactive terminal where the user selects an encryption type and initiates sensor data transmission.

### 5.4.2 MQTT Broker Operation (Mosquitto)

The Python script publishes encrypted sensor data to the 'sensor/data' topic using the MQTT protocol. The Mosquitto broker, operating on its default port 1883, facilitates the communication by forwarding messages from the Python publisher to the Node-RED subscriber. No modifications were made to the default Mosquitto configuration files for this setup.

```
jiahuey@Jias-MacBook-Air-4 sendMac % netstat -an | grep 1883
tcp4      0      0 127.0.0.1.1883      127.0.0.1.62580    ESTABLISHED
```

Figure 5.2: Verification of MQTT Broker Active

Figure 5.2 demonstrates that the Mosquitto MQTT broker is actively running on the local machine. Using the ‘netstat’ command in the terminal, the output confirms that port 1883 (default MQTT port) is open and an active TCP connection exists between the localhost (127.0.0.1) and a client. This indicates a successful connection and operation of the broker.

### 5.4.3 Receiving and Decrypting in Node-RED

Node-RED, running on the same machine or network, receives messages published to the MQTT broker. The process flow includes:

- The MQTT Input Node listens for incoming encryption packets.
- Encryption Type Function Node parses the message and determines the encryption type.
- A Switch Node uses the encryptionType variable and passes it to the correct decryption function.
- The corresponding function (AES, ChaCha20, Twofish) is executed using globally loaded libraries.
- The result is parsed into readable JSON format and passed downstream. The next Figure 5.3 highlights the full Node-RED flow.

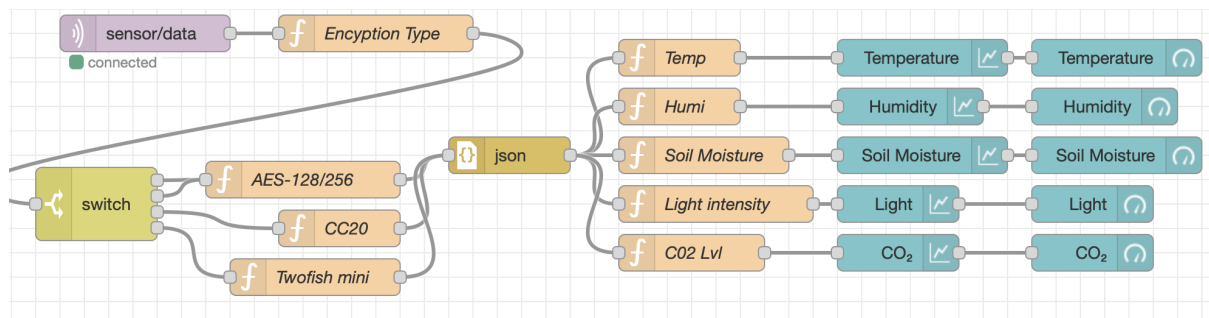


Figure 5.3: Node-RED Flow

Figure 5.3 highlights the flow logic and how the system automatically routes each message to its respective function node based on encryption type. It begins with MQTT-in node in purple, to Encryption Type function node in orange, to a switch node in yellow, splitting and heading into the appropriate decryption node in orange. Upon successful decryption, it parses its decrypted message into a JSON formatted string through the json node in orange, and finally visualised the data in a visual form in the graph node in blue.

### 5.4.4 Dashboard Visualisation

Once decrypted, the sensor values are displayed in real time using Node-RED's Dashboard UI. Each environmental metric (Temperature, Humidity, Soil Moisture, Light, CO<sub>2</sub>) is shown using both line charts and gauges for immediate interpretation. These charts auto-update with each new packet. Figure 5.4 will show the operational UI dashboard for users to infer data.

**Arrange temperature graph and gauge horizontally, humi graph and gauge horizontally etc**

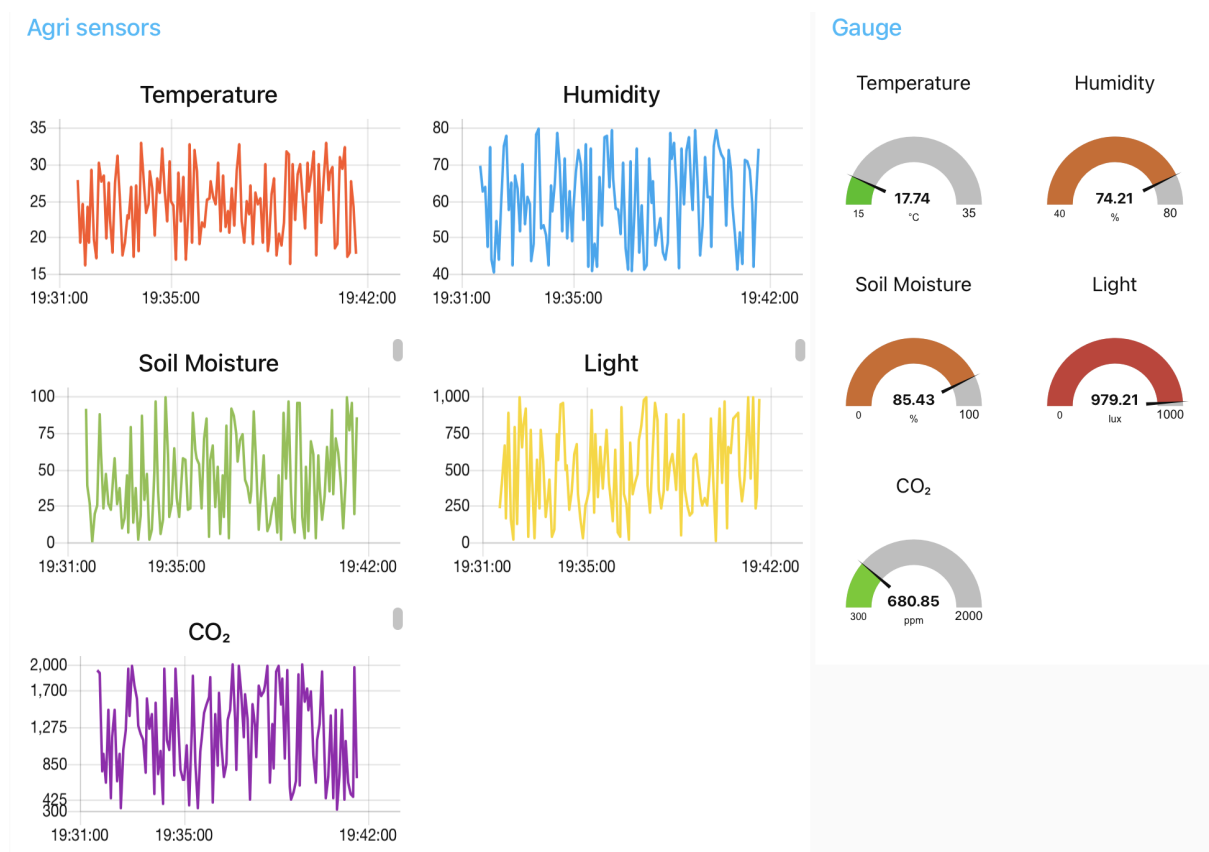


Figure 5.4: Node-RED Dashboard UI view

Figure 5.4 shows the dashboard live readings for each simulated sensor metric using charts and gauges, enabling easy monitoring of encrypted IoT data in real time.

### 5.4.5 Benchmark Script Execution Flow

In a separate workflow, a standalone benchmarking script is used to evaluate the performance of each encryption algorithm. This script is run independently of the MQTT transmission:

- Execution time
- CPU and memory utilization

- Throughput
- Decryption verification
- Energy consumption estimation

While not part of the continuous system, this component plays an important role in assessing which encryption mode is most appropriate under agricultural environments. Figure 5.5 will show a snippet of the benchmarking result.

```
=== Results for 1.0MB of high entropy data ===
```

Algorithm	Encrypt Time (s)	StdDev	Decrypt Time (s)	StdDev	Encrypt MB/s	Decrypt MB/s	CPU %	Energy Score
AES-128	0.003224	0.000138	0.006887	0.002611	302.93	141.81	58.8	0.3
AES-256	0.01069	0.001516	0.007859	0.003229	91.35	124.26	25.7	0.2
ChaCha20	0.019403	0.003929	0.027502	0.002826	50.33	35.51	21.9	0.5
Twofish	0.91786	0.016113	0.940852	0.017663	1.06	1.04	27.5	25.6

Figure 5.5: Benchmarking Script Result Demonstration

Figure 5.5 shows the sample result in the form of a screenshot of the completed benchmark script of putting all encryption under a benchmark with 1 MB high entropy string. It highlights the algorithm used, and all the variables that were tested. However a graph will be used in favor of this tabulated result. Based on this specific instance of the result, AES-256 appears to be the appropriate choice due to a balance of CPU usage, more analysis and discussion on Chapter 6.

## 5.5 Implementation Issues and Challenges

During the development and integration of the secure data transmission system, several technical challenges were encountered. These issues primarily stemmed from Node-RED's limitations in handling external cryptographic libraries and the complexity of implementing low-level ciphers such as Twofish.

### 5.5.1 Node-RED function Context Configuration

Node-RED function nodes by default lack access to external libraries. Attempting to import modules such as 'crypto.js', 'sodium-native', or 'twofish-ts' within function nodes returned errors related to undefined modules. This was resolved by explicitly defining the 'functionGlobalContext' in 'settings.js'. After saving the file, Node-RED had to be restarted for the context to apply. This approach made these libraries globally accessible across flows which allowed for decryption to work. This issue was particularly pronounced on macOS due to stricter default permissions in npm.



### 5.5.2 Block Cipher Padding Constraints

Since both AES and Twofish are block ciphers requiring fixed 16-byte input lengths, proper padding was essential. The project adopted **PKCS#7 padding**, a widely used scheme where padding bytes reflect the number of missing bytes needed to complete the block. Without this padding mechanism, ciphertext decryption would fail with alignment errors, especially when using AES ECB. Padding and unpadding were handled manually in both the encryption (Python) and decryption (Node-RED) side.

### 5.5.3 Complexity of Twofish Implementation in Node-RED

Integrating Twofish decryption in Node-RED posed a significant challenge due to the low-level API design of the 'twofish-ts' package. Unlike AES or ChaCha20 which offer streamlined encryption/decryption, twofish-ts requires manual handling of cipher blocks and session keys.

Key challenge included:

- **Manual block processing:** Ciphertext had to be decrypted in 16-byte chunks, requiring looped buffer operations.
- **Key derivation:** Keys were derived using SHA-256 hashing to meet byte-length constraints.
- **Padding validation:** Post-decryption, padding had to be validated and removed safely using PKCS#7 rules.

The need to manage session arrays (sBox and sKey) and align input/output buffers correctly made Twofish decryption notably more complex compared to the single-call decryption available in AES and ChaCha20's implementations.

### 5.6 Concluding Remark

This chapter has detailed the complete implementation process of the secure IoT data transmission system, from hardware and software setup to configuration and runtime operation. Through a modular Python-based sender script and a dynamic Node-RED receiver flow, the system demonstrated real-time encryption, transmission, and decryption of simulated agricultural sensor data. Despite development being conducted largely in a simulated environment, the design ensured compatibility with physical Raspberry Pi deployment and hardware extensions.

## Chapter 6

# System Evaluation and Discussion

### 6.1 System Testing and Performance Metrics

The performance of the proposed encryption-based data transmission system was evaluated to determine its efficiency, responsiveness, and resource usage under various operating conditions. These evaluations were designed to simulate real-world smart agriculture deployments, particularly in low-power or embedded IoT environments such as the Raspberry Pi Zero 2 W.

Testing was conducted on both the development machine (MacBook Air M1) and the target hardware (Raspberry Pi Zero 2 W). All four encryption modes—AES-128, AES-256, ChaCha20, and Twofish—were included in the benchmark process using a custom benchmarking script written in Python. Results and observations are presented in Section 6.2 through a series of comparative visual analysis.

#### 6.1.1 Objectives of Testing

The key goals of the system testing were:

- To assess the **execution time** of encryption and decryption across algorithms.
- To evaluate **system throughput**, i.e., the rate at which data could be securely processed.
- To monitor **CPU load and memory usage**, indicating the efficiency of each algorithm.
- To estimate **energy consumption**, derived from CPU usage and operational duration, critical for edge device sustainability.

#### 6.1.2 Test Parameters and Methodology

The benchmarking script was developed in Python and executed under consistent conditions on both platforms. Each encryption algorithm was tested using randomized input data of varying sizes and entropy to simulate realistic transmission payloads.

##### Data Sizes Tested:

- 1 KB
- 10 KB

- 100 KB
- 1 MB
- 4.9 MB
- 5 MB and 10 MB (Optional, may crash or become impractically slow for Twofish)

Due to the computational overhead of Twofish, particularly on the Raspberry Pi, the benchmarking script includes a prompt for the user to proceed when evaluating 4.9MB and above. In some cases, execution exceeded 30 minutes or caused system instability, resulting in the omission of those data points in the analysis.

### 6.1.3 Entropy Conditions:

- **High entropy** (random binary data) - ‘abcdef’
- **Low entropy** (repetitive or compressible data) - ‘aaaaaa’

### 6.1.4 Repetitions:

- Each scenario was executed three times to ensure statistical relevance and smooth out anomalies. The result is then averaged out as one output and will be used as the point of reference for the analysis.

### 6.1.5 Performance Metric Tracked

Table 6.1 will discuss the metrics that will be used for analysing the benchmark results.

Table 6.1: Metrics Tracked and Description

Metric	Description
Encryption Time (ms)	Time taken to encrypt payload
Decryption Time (ms)	Time to recover the original data
Throughput (MB/s)	Amount of data securely processed per second
CPU Usage (%)	Average CPU load during operation
Memory Usage (KB)	Peak memory allocated
Energy Estimation	Indirect metric based on CPU load over time

Table 6.1 lists the performance metrics considered when analysing the benchmarking results. All measurements were collected using Python libraries such as ‘psutil’, ‘time’, and built-in

statistical modules. Result sets were tabulated using ‘tabulate’ and visualised using ‘matplotlib’

This testing framework provides the basis for the comparative performance evaluations presented in Section 6.2, encompassing efficiency, scalability, and hardware suitability across all algorithms and platforms.

## 6.2 Testing Setup and Results

This section will prove the evaluation results of the encryption system’s performance, as benchmarked on both a Macbook Air M1 and the Raspberry Pi Zero 2 W. Each encryption algorithm (AES-128, AES-256, ChaCha20, and Twofish) was tested across different payload sizes (1 KB, 10 KB, 100 KB, 1.0 MB, and 4.9 MB). The results are visualised to highlight differences in encryption efficiency, throughput, and resource utilisation.

### 6.2.1 Encryption Time

Figure 6.1 illustrates the encryption times required by each algorithm at different data sizes for both the Macbook and Raspberry Pi.

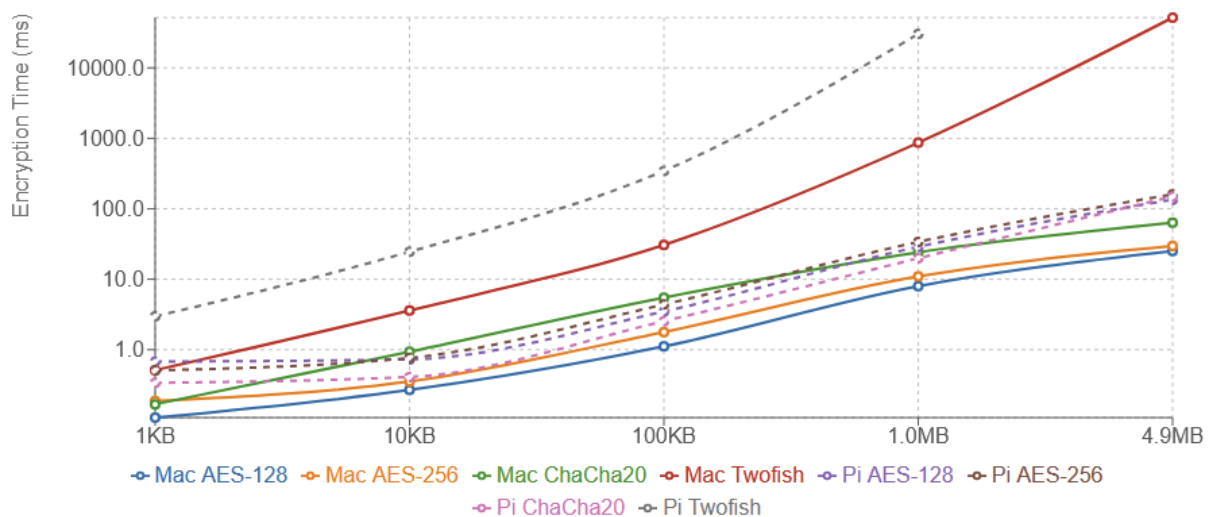


Figure 6.1: Encryption Time Comparison Across Algorithms and Devices (Lower is Better)

Figure 6.1 shows the encryption time performance of AES-128, AES-256, ChaCha20, and Twofish across varying data sizes (1 KB, 10 KB, 100 KB, 1.0 MB, 4.9 MB) tested on both MacBook Air M1 and Raspberry Pi Zero 2 W. The graph visualises the time required for each encryption operation, with notable differences in performance depending on the algorithm

and hardware platform. AES-128 consistently demonstrates the lowest encryption time across all payload sizes, making it the most efficient algorithm in terms of speed. AES-256 follows closely behind but incurs slightly higher computational costs due to the extended key length. ChaCha20, while slightly slower than AES-128 and AES-256, maintains relatively consistent performance, especially on the Raspberry Pi. In contrast, Twofish displays significantly higher encryption times, particularly for larger payloads such as 1MB and 4.9MB, where its performance degrades dramatically. This issue is especially critical on the Raspberry Pi, where Twofish encryption time exceeds practical limits for real-time IoT operations. Twofish is an outlier due to its more complex key schedule and computational overhead, which heavily burden low-power devices like the Raspberry Pi. The findings indicate that lightweight algorithms like AES-128 are far more suitable for resource-constrained environments in smart agriculture deployments, while Twofish's computational overhead renders it less viable for such use cases. The next figure (Figure 6.2) will discuss the decryption time of the encryption algorithm.

### 6.2.2 Decryption Time

Figure 6.2 illustrates the decryption times required by each algorithm at different data sizes for both the Macbook and Raspberry Pi.

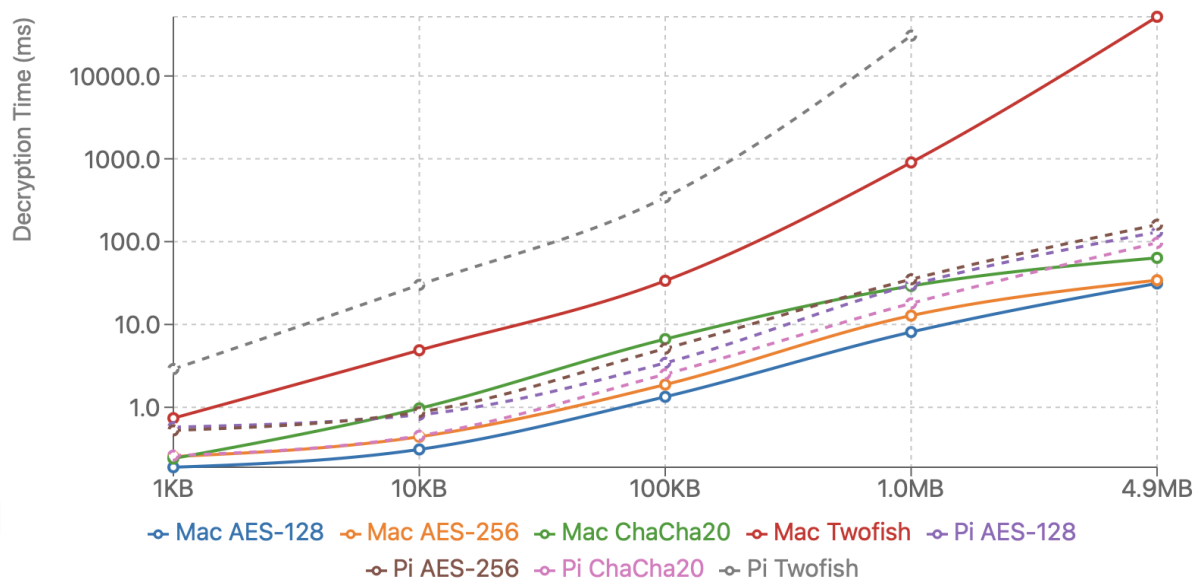


Figure 6.2: Decryption Time Comparison Across Algorithms and Devices (Lower is Better)

Figure 6.2 illustrates the decryption time performance of AES-128, AES-256, ChaCha20, and Twofish across different payload sizes on both Macbook Air M1 and Raspberry Pi Zero 2 W.

Decryption time patterns largely mirror encryption performance trends observed previously in Figure 6.1. AES-128 consistently offers the fastest decryption times across all tested data sizes, confirming its suitability for real-time smart agriculture data handling. AES-256 follows closely but imposes slightly more computational overhead due to its extended key size. ChaCha20 performs moderately, exhibiting competitive speeds for small to medium payloads, particularly on the Raspberry Pi, where it even outperforms AES encryption at 4.9MB scale. Conversely, Twofish again shows extremely high decryption times, especially for payloads exceeding 1MB, where execution becomes impractically slow, notably on the Raspberry Pi. The decryption of a 1MB file using Twofish takes approximately 30696 ms (around 30 seconds), and the 4.9MB test is practically infeasible. Twofish is an outlier because its decryption process involves intensive mathematical operations and S-box lookups that are inefficient on constrained hardware.

The logarithmic scale representation is necessary to accommodate the extremely disproportionate behaviour of Twofish relative to the other algorithms. These results further reinforce that Twofish is unsuitable for lightweight, latency-sensitive applications in IoT environments, while AES-128, AES-256, and ChaCha20 remain strong candidates depending on specific system constraints. Figure 6.3 will touch on the encryption throughput of the algorithms.

### 6.2.3 Encryption Throughput

Figure 6.3 shows the encryption throughput performance for AES-128, AES-256, ChaCha20, and Twofish algorithms across varying data sizes on both the MacBook Air M1 and Raspberry Pi Zero 2 W platforms.

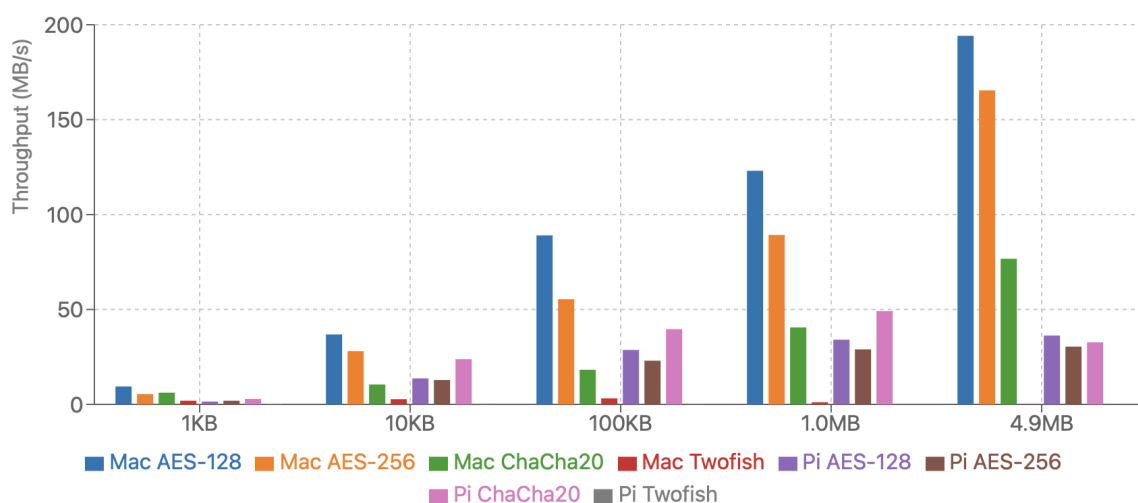


Figure 6.3: Encryption Throughput Comparison Across Mac and Raspberry Pi (Higher is Better)

Figure 6.3 illustrates the encryption throughput achieved by each algorithm across varying data sizes for both the MacBook Air M1 and Raspberry Pi Zero 2 W platforms. As expected, the MacBook consistently outperformed the Raspberry Pi in terms of throughput due to its more powerful hardware capabilities. Among all algorithms, AES-128 demonstrated the highest throughput across all data sizes on both devices, reaching approximately 194 MB/s on the Mac and about 36 MB/s on the Pi for a 4.9MB payload. AES-256 followed closely behind but showed slightly reduced performance compared to AES-128, reflecting the additional computational overhead required by its longer key length.

ChaCha20 also exhibited competitive throughput, particularly on the Raspberry Pi, where it occasionally outperformed AES-256 at specific payload sizes. This outcome supports ChaCha20's reputation for being efficient on low-power, embedded devices. Meanwhile, Twofish displayed the lowest throughput among all algorithms, remaining below 1 MB/s on both Mac and Pi even for small data sizes. This poor performance can be attributed to the lower-level nature of the 'twofish-ts' library used, which requires manual block-by-block decryption, adding significant computational delay. The gap between Mac and Pi was more pronounced for larger payloads, emphasizing the importance of selecting a lightweight and efficient encryption algorithm in smart agriculture deployments. Figure 6.4 will touch on the decryption throughput side of the benchmark.

### 6.2.4 Decryption Throughput

Figure 6.4 shows the decryption throughput performance for AES-128, AES-256, ChaCha20, and Twofish algorithms across varying data sizes on both the MacBook Air M1 and Raspberry Pi Zero 2 W platforms.

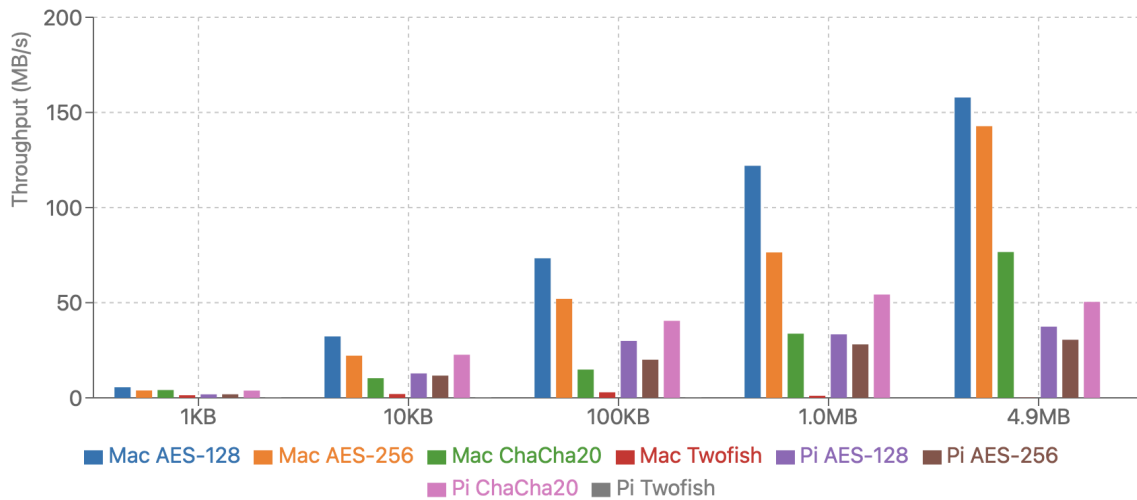


Figure 6.4: Decryption Throughput Comparison Across Mac and Raspberry Pi (Higher is Better)

Figure 6.4 presents the decryption throughput achieved by each encryption algorithm across various data sizes for both the MacBook Air M1 and Raspberry Pi Zero 2 W. As with encryption throughput, the MacBook consistently delivered superior performance compared to the Raspberry Pi. AES-128 again dominated in throughput performance across most tested payloads, achieving up to 157 MB/s for 4.9MB payloads on the Mac, and around 37 MB/s on the Pi. AES-256 followed but lagged slightly due to the additional computational overhead of its longer key size, especially noticeable for larger payloads.

ChaCha20 displayed a particularly strong showing during decryption tests, especially on the Raspberry Pi. Notably, it surpassed AES-256 throughput at several data points, reinforcing ChaCha20's reputation for efficiency on lower-power devices. This observation is important for real-world IoT deployments where decryption speed can impact responsiveness of edge applications. Twofish once again recorded the poorest performance, with extremely low throughput values, especially evident for larger payloads where its figures dropped below 1 MB/s or were not measurable within reasonable time frames. Twofish is the outlier because its decryption relies on CPU-intensive routines poorly suited for real-time performance on constrained devices. This substantial gap in Twofish's decryption performance compared to



other algorithms reinforces the earlier concerns regarding its unsuitability for lightweight IoT platforms like Raspberry Pi. Figure 6.5 will discuss the memory usage of the algorithms.

### 6.2.5 Memory Usage

Figure 6.5a and Figure 6.5b show the memory usage of AES-128, AES-256, ChaCha20, and Twofish algorithms across varying data sizes on both the MacBook Air M1 and Raspberry Pi Zero 2 W platforms.

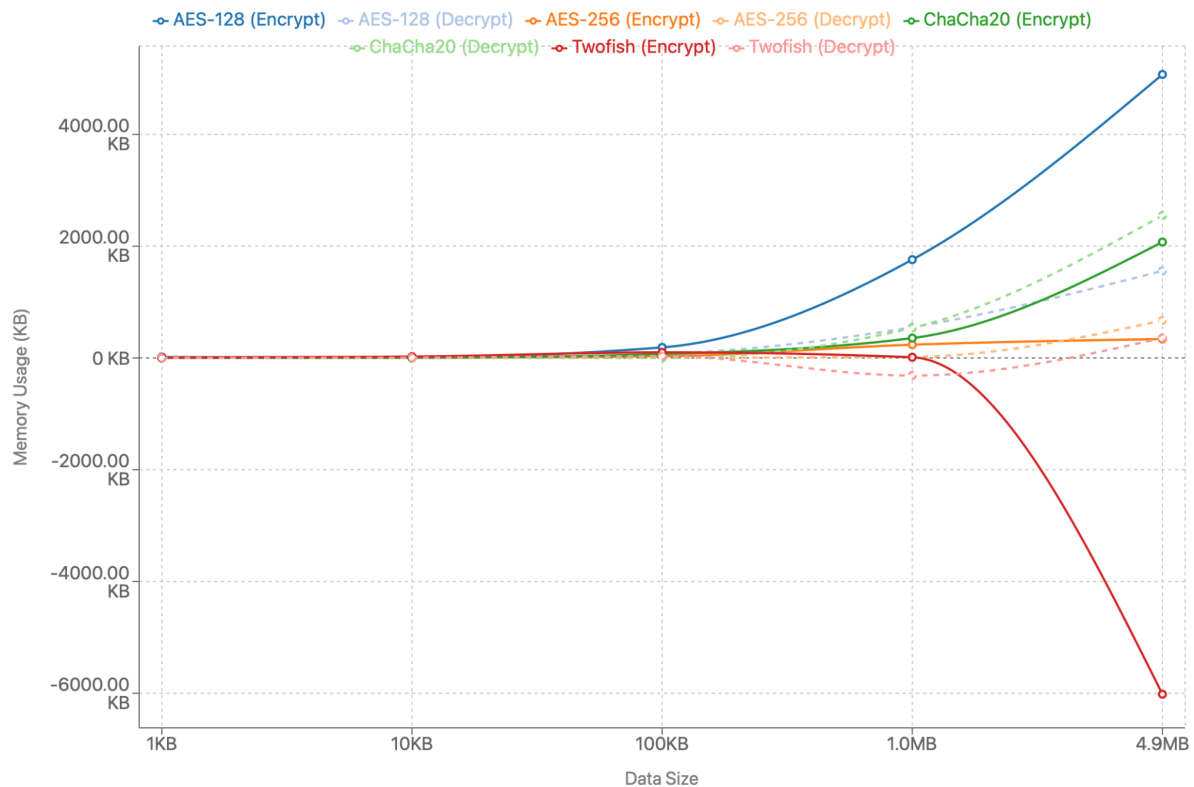


Figure 6.5a: Memory Usage of Encryption Algorithms on MacBook

Figure 6.5a shows the memory usage profile of AES-128, AES-256, ChaCha20, and Twofish during encryption and decryption operations on the MacBook Air M1. As the payload size increased, memory usage for AES-128, AES-256, and ChaCha20 generally scaled upward in a predictable manner. Twofish, however, exhibited erratic memory behaviour, including negative values at larger data sizes, most notably at 4.9MB where encryption memory dipped to -6017.07 KB. This anomaly is attributed to transient memory management optimizations or measurement artifacts in Python's 'psutil' memory tracking, particularly during long-running Twofish processes. Overall, AES and ChaCha20 maintained a lightweight and consistent memory footprint suitable for resource-sensitive applications. Figure 6.5b will discuss memory usage on the Raspberry Pi instead.

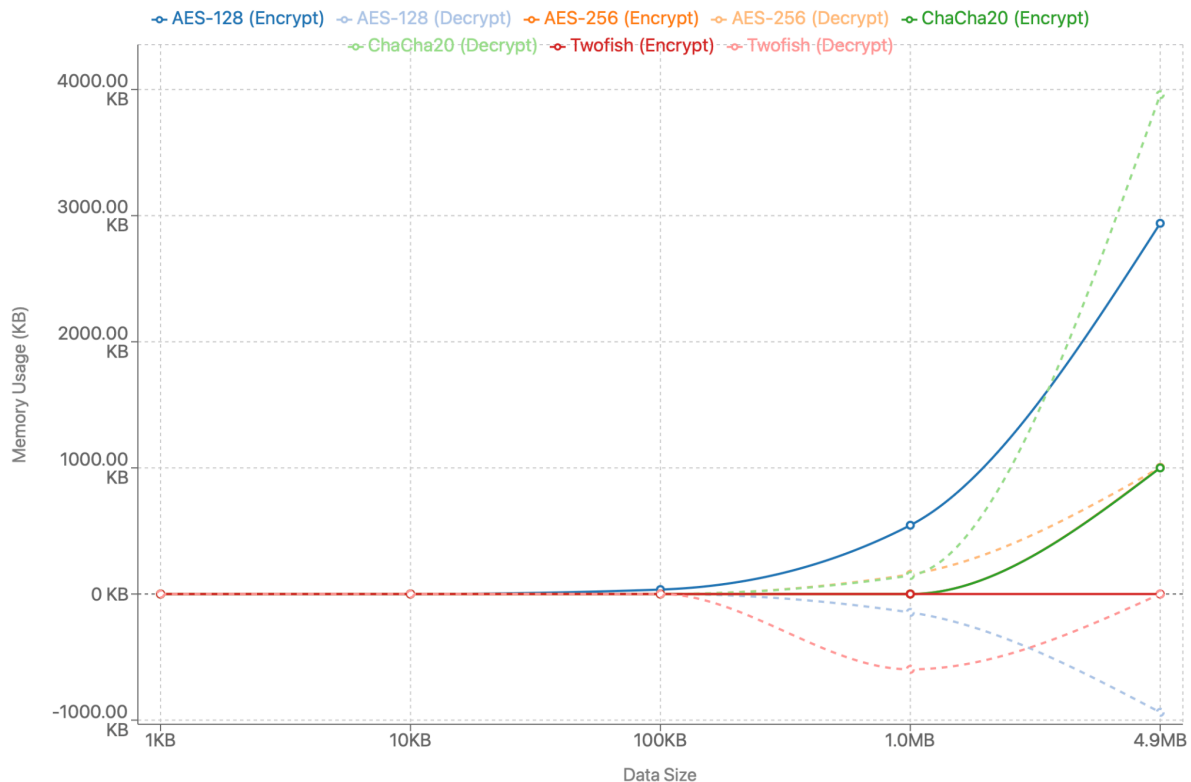


Figure 6.5b: Memory Usage of Encryption Algorithms on Raspberry Pi

Figure 6.5b displays the memory usage of encryption and decryption operations on the Raspberry Pi Zero 2 W. Consistent with the MacBook findings, AES-128, AES-256, and ChaCha20 demonstrated low and scalable memory consumption across increasing data sizes. Twofish again showed erratic behaviour, recording 0 KB or negative memory values during large data decryptions, such as -938.4 KB for AES-128 decryption of 4.9MB payloads. These anomalies likely result from Python's memory handling combined with aggressive garbage collection on resource-constrained systems like the Pi. Nonetheless, AES and ChaCha20 maintained superior stability, further reinforcing their suitability for low-memory embedded devices compared to the resource-intensive Twofish algorithm.

Across both the MacBook Air M1 and Raspberry Pi Zero 2 W, the encryption and decryption memory usage patterns reaffirm the lightweight nature of AES and ChaCha20 encryption, making them highly suitable for resource-constrained IoT devices. Meanwhile, Twofish presented irregular memory behaviours, particularly under large payload conditions, which complicates its deployment in memory-sensitive environments. These findings highlight the importance of memory efficiency alongside speed and security when selecting cryptographic

algorithms for agricultural IoT systems. The next section will discuss CPU Usage across all algorithms on both hardware.

### 6.2.6 CPU Usage

This section examines the CPU utilization incurred by each encryption and decryption process on both the Mac and Raspberry Pi platforms with Figure 6.6a and Figure 6.6b respectively. Two representative data sizes — 1MB and 4.9MB — are selected to reflect medium and large data payloads common in smart agriculture systems. The CPU usage was recorded as the average load over the duration of each operation using Python's 'psutil' module. Algorithms tested include AES-128, AES-256, ChaCha20, and Twofish.

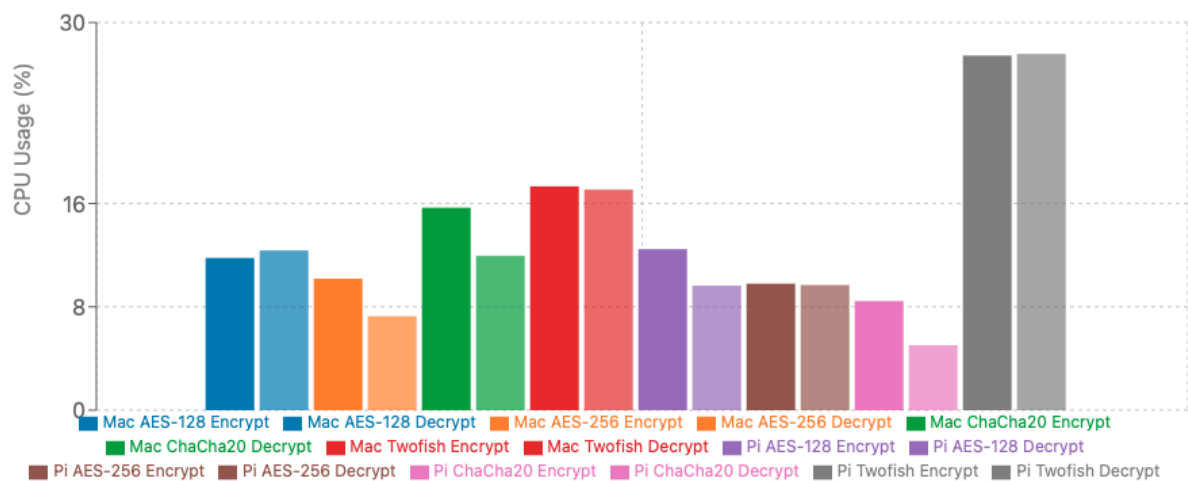


Figure 6.6a: CPU Usage per Encryption/Decryption Operation at 1MB on Mac and Pi (Lower is Better)

Figure 6.6a compares CPU usage for encrypting and decrypting 1MB data chunks. On the Mac, AES-128 and AES-256 performed efficiently, consuming under 12% CPU, while ChaCha20 showed slightly higher usage (~15.67% encryption). Twofish registered the highest usage (~17.3%) among all algorithms. On the Raspberry Pi, Twofish again recorded significantly higher load (27.4%), while AES and ChaCha20 remained under 13%, indicating their suitability for edge deployment.

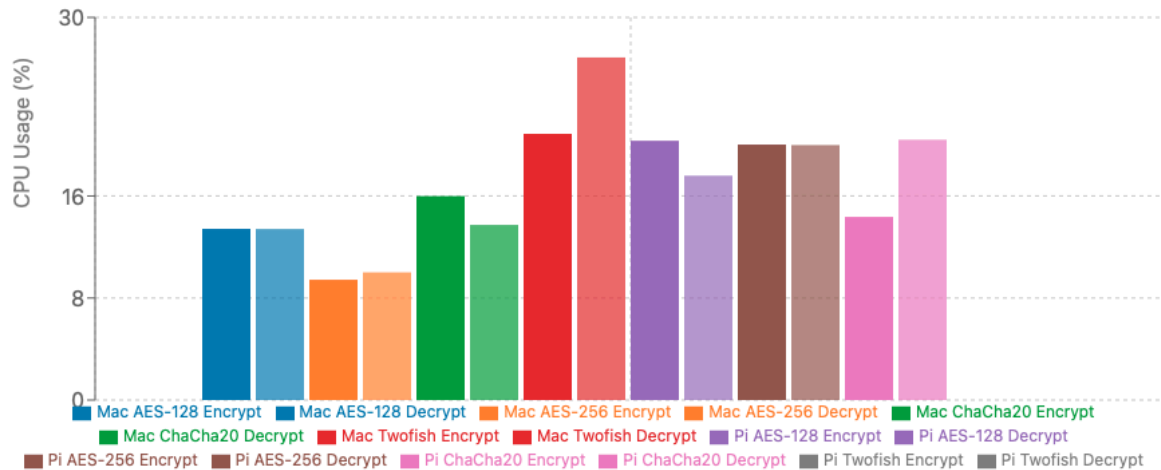


Figure 6.6b: CPU Usage per Encryption/Decryption Operation at 4.9MB on Mac and Pi (Lower is Better)

Figure 6.6b explains CPU usage for encrypting and decrypting 4.9MB data chunks. Twofish was the most CPU-intensive on the Mac, reaching ~26.8%, but no CPU data was recorded for Twofish on the Raspberry Pi, likely due to memory exhaustion or process failure as discussed in Section 6.2.5. AES-256 and ChaCha20 remained relatively stable, indicating better scalability for edge computing applications.

Overall, CPU usage trends demonstrate that lightweight ciphers such as AES-128, AES-256, and ChaCha20 are well-suited for both development and edge devices, maintaining reasonable load even at larger data sizes. Twofish consistently exhibited the highest CPU consumption, particularly on the Raspberry Pi where it may even fail, highlighting its inefficiency for constrained hardware. These findings are crucial when selecting encryption schemes for IoT applications where processing power and battery life are limited. Figure 6.7 will discuss the energy estimation of the encryption algorithms.

### 6.2.7 Energy Estimation

Energy consumption was indirectly assessed through a relative energy score derived from CPU usage and execution time. This metric is essential in evaluating the sustainability of cryptographic operations in resource-constrained devices like the Raspberry Pi, especially in long-term agricultural deployments where energy efficiency directly affects operational viability. Figure 6.7a will address the energy estimation at 1MB.

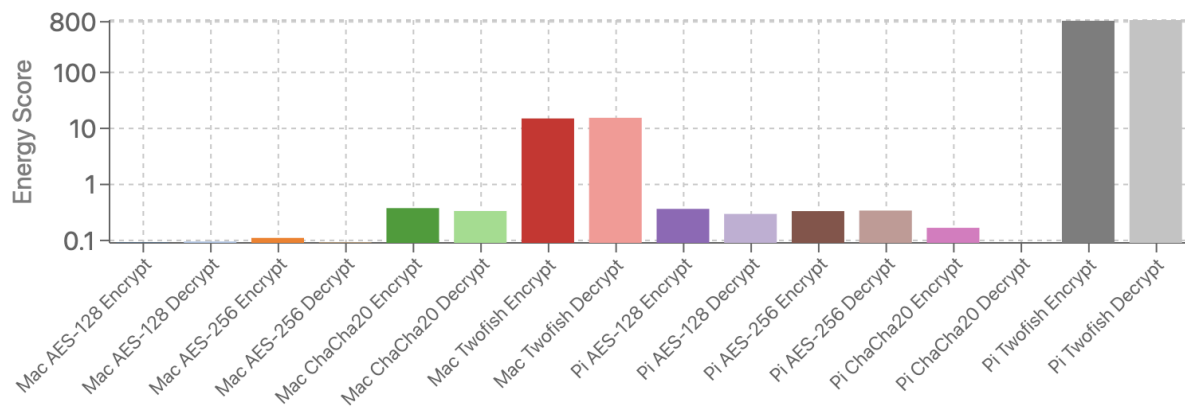


Figure 6.7a: Energy Estimation at 1MB Data Size for Mac and Pi (Lower is better)

Figure 6.7a illustrates the estimated energy score for encryption and decryption of a 1MB payload across all algorithms and platforms. Notably, the MacBook maintained consistently lower energy profiles across most algorithms, with AES-128 and AES-256 presenting the lowest scores overall. In contrast, ChaCha20 exhibited moderately higher energy use on the Mac, while Twofish remained significantly more demanding across both platforms. The Raspberry Pi's energy scores were elevated compared to the Mac, especially for Twofish, where both encryption and decryption registered exponentially higher values relative to other algorithms — a behaviour consistent with previous performance bottlenecks. Figure 6.7b will switch the data size into 4.9MB.

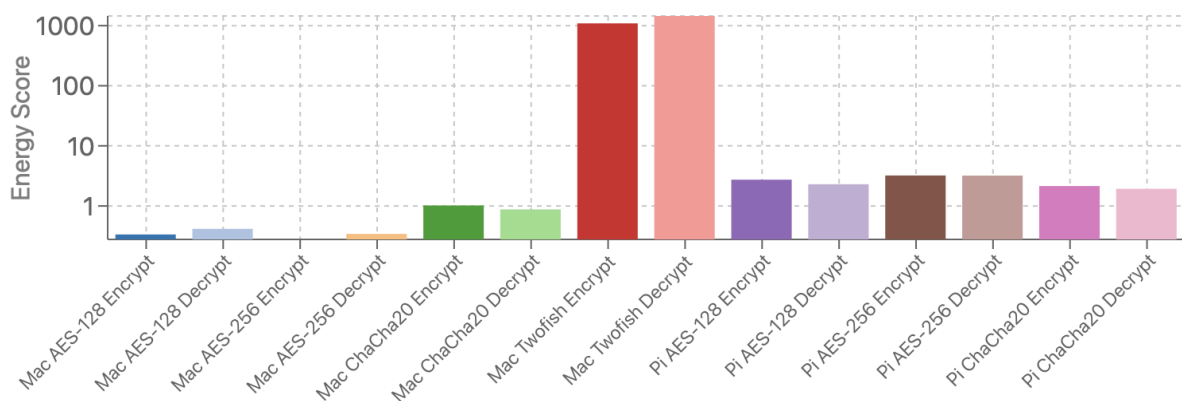


Figure 6.7b: Energy Estimation at 4.9MB Data Size for Mac and Pi (Lower is better)

Figure 6.7b expands the comparison to 4.9MB data size, where energy disparities become even more pronounced. Twofish on the Mac recorded energy scores exceeding 1000 units, highlighting its unsuitability for large data encryption in constrained environments. On the Raspberry Pi, energy scores for AES and ChaCha20 remained within a manageable range. However, Twofish failed to complete the 4.9MB operation on the Pi, leading to its omission from the graph — an outcome that reinforces previously observed memory and CPU limitations (see Figures 6.4b and 6.5b). ChaCha20 again proved to be a competitive option, balancing speed and energy efficiency effectively, especially in decryption.

The energy estimation results strongly suggest that AES-128 and ChaCha20 offer the most favorable balance between performance and energy efficiency across both platforms. Twofish, while theoretically secure, exhibited extreme power demands and operational instability on low-power devices, undermining its feasibility for real-world deployments in smart agriculture IoT networks.

### **6.3 Project Challenges**

During the development and benchmarking phases of this project, several challenges emerged that affected both the evaluation process and the overall system design. These challenges were primarily related to hardware limitations, library constraints, and inconsistencies in performance measurement. Each of these issues is discussed below to provide a realistic assessment of the project's technical hurdles.

#### **6.3.1 Performance Bottlenecks with Twofish**

Twofish consistently underperformed in all benchmark metrics, including encryption/decryption time, throughput, CPU usage, memory consumption, and energy estimation. These issues were especially pronounced on the Raspberry Pi Zero 2 W, where the algorithm either failed to complete at larger payloads (e.g., 4.9MB) or returned extreme results—such as over 1000 in relative energy score and null outputs.

This performance degradation can be attributed to the limitations of the ‘twofish-ts’ Python library used. Unlike AES and ChaCha20, which benefit from optimised, low-level libraries with hardware acceleration (e.g., OpenSSL-backed bindings), Twofish required manual

block-by-block encryption logic. This approach led to significantly higher computational overhead, making the algorithm unsuitable for lightweight real-time IoT applications.

### 6.3.2 Resource Constraints on the Raspberry Pi

Although the Raspberry Pi Zero 2 W features a quad-core CPU, the benchmark script was written in a single-threaded Python process and could not effectively utilise multiple cores. As a result, computational load was limited by weak single-core performance, particularly when dealing with large data sizes or complex algorithms like Twofish.

This limitation led to excessive CPU usage, prolonged execution times, and in some cases, incomplete benchmarks. While lightweight algorithms like AES-128 and ChaCha20 performed acceptably, Twofish repeatedly caused operational instability, demonstrating the need to match algorithm complexity with the processing capabilities of the target hardware.

### 6.3.3 Memory and Energy Measurement Anomalies

Another significant challenge was the inconsistency in memory usage reporting. During large-payload encryption or decryption tasks, especially with Twofish, the ‘psutil’ library recorded negative memory usage or 0 KB values. This anomaly likely resulted from aggressive garbage collection or memory reuse strategies by Python’s runtime, especially under constrained environments like the Raspberry Pi.

Similarly, the energy estimation metric—derived from CPU usage and operation time—exhibited significant variance. Though useful as a relative indicator, its accuracy was inherently limited by the software-based approximation method and the non-deterministic nature of process scheduling in Python.

### 6.3.4 Library Fragmentation and Tool Limitations

The benchmarking process faced additional complexity due to differences in how encryption algorithms are implemented across libraries. For example, AES encryption was available via well-optimised libraries (such as ‘cryptography’), whereas ChaCha20 required use of ‘PyCryptodome’, and ‘Twofish’ depended on the less-performant ‘twofish-ts’.

This fragmentation meant that no single library could be used uniformly across all algorithms. Consequently, special handling was needed in the benchmarking script for each

Bachelor of Information Technology (Honours) Communications and Networking  
Faculty of Information and Communication Technology (Kampar Campus), UTAR

algorithm, which introduced small but unavoidable disparities in how performance was measured.

### **6.3.5 Manual Interruption Handling for Long-running Tests**

To prevent prolonged execution or system crashes during testing, the benchmarking script included an interactive prompt asking the user whether to continue with Twofish evaluations for 4.9MB, 5MB, and 10MB payloads. While this approach safeguarded system stability, it disrupted the flow of automated benchmarking and required human intervention, which is not ideal for repeatable experimentation.

### **6.4 Objectives Evaluation**

To reiterate, the objective of this project is to develop a secure and replicable data transmission system for smart agriculture in Malaysia using Raspberry Pi and Node-RED. The system must transmit simulated sensor data using MQTT while implementing encryption algorithms – AES-128, AES-256, ChaCha20, and Twofish to assess their performance in a low-power, IoT-based environment. The system also aims to identify an algorithm that provides a practical balance of speed, resource efficiency, and security for real-world developments.

Upon evaluation, the project has successfully met its objectives. The encryption system was implemented and deployed using both high-end (Macbook Air M1) and low-power (Raspberry Pi Zero 2 W) platforms. All four algorithms were integrated and benchmarked across multiple data sizes with measurements taken for encryption time, decryption time, throughput, memory usage, CPU load, and energy estimation. AES-128 emerged as the most well-rounded and optimal algorithm, offering the best trade-off between speed, resource efficiency, and energy use—particularly on the Raspberry Pi. ChaCha20 also performed exceptionally well, especially in memory-constrained scenarios, making it a strong alternative for certain IoT deployments. In contrast, Twofish displayed critical weaknesses in processing speed, memory handling, and energy use, disqualifying it as a practical option for constrained environments. The system, built entirely with open-source tools, remains low-cost, replicable, and suitable for future enhancements.



### 6.5 Concluding Remark

This chapter presented a comprehensive evaluation of the implemented secure data transmission system for smart agriculture, highlighting its performance across multiple cryptographic algorithms on both a development machine (MacBook Air M1) and a target embedded device (Raspberry Pi Zero 2 W). Through rigorous testing, AES-128 emerged as the most efficient and reliable algorithm overall, delivering the best balance of performance and energy for real-time IoT agricultural applications. ChaCha20 closely followed, demonstrating solid efficiency and scalability on low-power hardware, making it a strong second choice depending on specific constraints.

In contrast, Twofish exhibited significant performance limitations especially on CPU and memory metrics, rendering it unsuitable for constrained edge platforms like the Raspberry Pi. The benchmarking effort highlighted the critical role of lightweight cryptography in resource-constrained environments, validating the project's core objective. Overall, this system is ready to support future smart agriculture initiatives with secure, efficient, and scalable data communication.

## Chapter 7

### Conclusion and Recommendations

This project successfully achieved its primary objective of developing, implementing and evaluating a secure data transmission system tailored for smart agriculture applications. By integrating four widely recognised symmetric encryption algorithms (AES-128, AES-256, ChaCha20, and Twofish), the system provided a versatile platform for assessing cryptographic performance in both development (MacBook Air M1) and embedded (Raspberry Pi Zero 2 W) environments.

Comprehensive benchmarking tests were conducted to evaluate encryption and decryption time, throughput, CPU usage, memory usage, and energy estimation across different payload sizes. The results showed that AES-128 consistently delivered the most favorable balance of speed, efficiency, and energy consumption, making it the definitive choice for low-power IoT systems. ChaCha20, while slightly less efficient in raw speed, demonstrated strong suitability for embedded applications, thanks to its low memory and CPU demands. AES-256, while secure, imposed greater computational overhead and latency, making it better suited for systems prioritising high security over speed.

Conversely, Twofish, despite its theoretical security strengths, underperformed across all tested parameters—particularly in large payload scenarios due to high latency, energy demands, and memory irregularities.

These findings highlight the need for encryption algorithm selection in IoT deployments to go beyond just cryptographic strength. Real-world considerations such as processing overhead, energy use, and platform compatibility are crucial. In summary, AES-128 stands out as the best-fit algorithm, with ChaCha20 offering an excellent alternative where performance trade-offs are acceptable.

#### 7.2 Recommendation

Based on the evaluation and findings of this study, several improvements are recommended for future development and research. First, the underperformance of the Twofish algorithm may be attributed to the limitations of the ‘twofish’ library used in the Node-RED and Python

environment. It is suggested that a more optimised or native implementation in a lower-level language such as C be considered, interfaced via Python using libraries like ‘ctypes’ or ‘cffi’ to yield more representative results.

Additionally, the benchmarking script employed in this project likely utilises a single CPU thread, which may not fully capitalize on the Raspberry Pi Zero 2 W’s quad-core architecture. Optimising the script to support multithreading could improve test realism and better reflect performance potential. Future versions of this system should also incorporate physical power measurement tools to obtain more accurate energy consumption data, as the current energy estimation is based on CPU load and execution time only.

Moreover, real-world testing in a live agriculture deployment is encouraged. This would introduce practical variables such as sensor noise, wireless transmission delays, and long-term stability that are difficult to simulate in controlled environments. A deeper examination of how input data entropy influences algorithm performance is also recommended, especially since agricultural sensor data may exhibit predictable or repetitive patterns that affect compression and encryption behaviour.

Finally, future studies could broaden the algorithm selection to include lightweight cryptographic algorithms such as Speck, Simon, or Ascon, which has been considered in NIST’s Lightweight Cryptography competition. These algorithms may offer better performance for embedded IoT use cases. Together, these recommendations aim to enhance the robustness, efficiency, and applicability of secure data transmission systems in real-world smart agriculture deployments.

## REFERENCES

- [1] A. Kamilaris, A. Kartakoullis, and F. X. Prenafeta-Boldú, "A review on the practice of big data analysis in agriculture," *Computers and Electronics in Agriculture*, vol. 143, pp. 23-37, 2017.
- [2] K. R. Pasupuleti, S. Ramalingam, and P. M. Kumar, "An efficient and secure smart agriculture framework using blockchain technology," *Computers and Electronics in Agriculture*, vol. 171, p. 105338, 2020.
- [3] Y. Zhang, R. Yu, S. Xie, W. Yao, and Y. Ming, "An improved data transmission security protocol for wireless sensor network and IoT," *Journal of Network and Computer Applications*, vol. 119, pp. 1-7, 2018.
- [4] K. O. Flores, I. M. Butaslac, J. E. M. Gonzales, S. M. G. Dumlao, and R. S. J. Reyes, "Precision agriculture monitoring system using wireless sensor network and Raspberry Pi local server," in *Proc. IEEE Region 10 Conf. (TENCON)*, Singapore, Nov. 2016, pp. 3018–3021.
- [5] S. Sontowski, M. Gupta, S. S. L. Chukkapalli, M. Abdelsalam, S. Mittal, A. Joshi, and R. Sandhu, "Cyber attacks on smart farming infrastructure," in *Proc. 2020 IEEE 6th Int. Conf. Collaboration Internet Computing (CIC)*, Oct. 2020, pp. 135–143.
- [6] V. Navalino, A. F. Wadjdi, and Y. Asnar, "Securing the Internet of Battlefield Things with ChaCha20-Poly1305 Encryption Architecture for Resource-Constrained Devices," *Int. J. Progressive Sci. Technol.*, vol. 42, no. 2, pp. 547–555, Jan. 2024.
- [7] A. Fotovvat, G. M. E. Rahman, S. S. Vedaiei, and K. Wahid, "Comparative performance analysis of lightweight cryptography algorithms for IoT sensor nodes," *IEEE Internet Things J.*, vol. 8, no. 10, pp. 8279–8290, Oct. 2021.
- [8] National Institute of Standards and Technology, *Specification for the Advanced Encryption Standard (AES)*, FIPS PUB 197, Nov. 2001.
- [9] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF protocols," RFC 7539, IRTF Crypto Forum, May 2015.


## References

- [10] B. Schneier, J. Kelsey, D. Whiting, N. Ferguson, D. Wagner, and C. Hall, *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*, John Wiley & Sons, 1999.
- [11] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.
- [12] M. N. Khan, A. Rao, and S. Camtepe, “Lightweight cryptographic protocols for IoT-constrained devices: A survey,” *IEEE Internet Things J.*, vol. 8, no. 5, pp. 4132–4156, 2020.
- [13] S. Blanc, A. Lahmadi, K. Le Gouguec, M. Minier, and L. Sleem, “Benchmarking of lightweight cryptographic algorithms for wireless IoT networks,” *Wireless Netw.*, vol. 28, pp. 3453–3476, Jul. 2022.
- [14] T. Hagino, *Practical Node-RED Programming: Learn Powerful Visual Programming Techniques and Best Practices for the Web and IoT*, Apress, 2016.
- [15] A. Banks and R. Gupta, *MQTT Version 3.1.1*, OASIS Standard, 29 Oct. 2014.
- [16] H. Eijs, *PyCryptodome 3.22.0 Documentation*, [Online]. Available: <https://www.pycryptodome.org>.
- [17] E. Upton and G. Halfacree, *Raspberry Pi User Guide*, 4th ed., Wiley, 2016.
- [18] Python Software Foundation, *Python 3.10.0 Documentation*, 2021. [Online]. Available: <https://docs.python.org/3>.

## POSTER

Faculty of Information Communication and Technology

**Secured Agriculture Sensor Data Based on end-to-end Encryption using Raspberry Pi**




**UTAR**  
UNIVERSITI TUNKU ABDUL RAHMAN

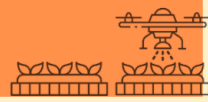
### Introduction

The **unencrypted transmission of sensitive data** collected from IoT sensors like **soil moisture, temperature, and nutrient levels** poses **significant security risks** in **providing real-time environmental data**.

### Objective



To design and implement a secure, cost effective data transmission system using Raspberry-Pi and Node-RED by incorporating end-to end encryption



To protect agricultural sensor data from unauthorized access and manipulation

### Proposed Method

- 1 Data Collection**  
 IoT sensors gather environmental data (soil moisture, temperature, humidity).
- 2 Data Transmission**  
 Data is sent to a Raspberry Pi and encrypted using a novel algorithm tailored for agricultural needs.
- 3 Data Processing**  
 The encrypted data is transmitted to an MQTT broker and then processed by Node-RED, which decrypts and visualizes the data on a dashboard.
- 4 Security**  
 The system ensures that all data transmissions are encrypted, maintaining data integrity and confidentiality.

### Why This System is Better

**Tailored Solution**

- Custom-built for agricultural data
- Outperforms generic encryption methods

**Cost Effective**

- Affordable with open source technologies
- Accessible for widespread adoption

**Enhanced Security**

- Incorporates robust encryption protocols for smart agriculture
- Ensure secure data transmission

### Conclusion

This project delivers a secure, efficient, and replicable system for smart agriculture in Malaysia, addressing the critical issue of unencrypted data transmission. The successful prototype lays the groundwork for a fully implemented system that will enhance the security and reliability of agricultural practices across the region.

Product Developer : Choo Jia Huey

Product Supervisor : Dr Abdulrahman Aminu Ghali