LONDON
METROPOLITAN
UNIVERSITY

Bio-Inspired Lightweight Polymorphic Security System for IoT Devices

Dion Miroy Mariyanayagam BEng(Hons) CEng MIET FHEA

A thesis is submitted in fulfilment of the requirements of London Metropolitan University for the degree of *Doctor of Philosophy*

April 2025

Abstract

Polymorphism, defined as the ability to dynamically alter form, has long been exploited by viruses and malware to evade traditional security mechanisms. This thesis proposes a novel application of polymorphic principles, inspired by biological immune systems, to engineer a lightweight, adaptive, and resilient security system for resource-constrained Internet of Things (IoT) devices. The Bio-Inspired Lightweight Polymorphic Security System introduces a comprehensive framework that detects, rejects, and neutralises unauthorised clients within a secure, encrypted client-server model. Drawing parallels to innate and adaptive immunity, the system dynamically rotates encryption keys, session credentials, and network configurations in real-time, ensuring robust defences against intrusion and desynchronisation threats.

Furthermore, the research identifies a critical limitation in conventional IoT security architectures: the lack of integrated, adaptive energy management. Addressing this gap, the thesis introduces the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS), a biologically inspired subsystem that predicts battery health trajectories and dynamically modulates operational states, ranging from active processing to ultra-low power sleep modes. The AABCMS mirrors biological neural energy management, adjusting system behaviour based on real-time energy availability to maximise device longevity without compromising security.

The entire system was implemented and validated on a custom ESP32-S3 development board and benchmarked against an ATMEGA328P microcontroller, encompassing extensive cycle, timing, power, and energy consumption analyses. Testing demonstrated the system's adaptive encryption selection, session integrity preservation during power fluctuations, desynchronisation recovery through honeypot redirection, and sustained security under energy-limited conditions.

The thesis concludes by situating this bio-inspired security architecture within broader technological trends, highlighting its potential synergy with machine learning, large language models (LLMs), and future quantum-resilient cryptographic methods. By uniting principles from immunology, embedded systems, cryptography, and adaptive energy management, this research contributes a pioneering interdisciplinary approach to the sustainable, secure, and autonomous evolution of IoT systems.

Acknowledgements

The journey from the beginning of this thesis has truly reflected my love and curiosity for engineering, and giving back to the engineering community has always been a heartfelt desire. This thesis was only achievable through my supervisors. Without the trust and support of Dr Pancham Shukla and Prof Bal Virdee, this research project would not have come to fruition. They have not only taken me under their wing but also allowed me to become a person of my own through this research, independent learning, and their advice. Both of my supervisors are the reason why people want to stay in higher education and pursue a postgraduate degree, as they are not just kind and welcoming but take time to talk to me weekly about my progress, it shows how much they love what they do, and it has immensely grown my respect and love for them. Finally, I would like to acknowledge the Edward Steers Memorial Fund for financially supporting the first year of my research.

Table of Contents

List of Tables

List of Figures and Illustrations

List of Abbreviations

Abbreviations	Definitions
AABCMS	Adaptive Amoeba Battery Curve Mapping Management System
AAC	Adaptive Amoeba Complexity
ACLP	Approved Clients List Process
ADC	Analogue-to-Digital Converter
ADS	Auto Detection System
AES	Auto Ejection System
AI	Artificial Intelligence
API	Application Programming Interface
AR	Augmented Reality
BFT	Byzantine Fault Tolerance
BLE	Bluetooth Low Energy
CBC	Cipher Block Chaining
CCL	Client Connection Logic
CFB	Cipher Feedback
CHF	Cryptographic Hash Function
CPU	Central Processing Unit
CTR	Counter
DAGs	Directed Acyclic Graphs
DQN	Deep Q Networks
ECB	Electronic Codebook
GDPR	General Data Protection Regulation
HAL	Hardware Abstraction Layer
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IIoT	Industrial Internet of Things
IoT	Internet of Things
LLC	Logical Link Control
LLM	Large Language Models
LP	Ledger Process
M2M	Machine-to-Machine
MAC	Media Access Control
ML	Machine Learning
MR	Mixed Reality
NIST	National Institute of Standards and Technology
NK	Natural Killer
NSA	National Security Agency
OFB	Output Feedback
OSI	Open Systems Interconnection
OTA	Over-The-Air
OWASP	Open Worldwide Application Security Project
PaaS	Platform as a Service
PBFT	Practical Byzantine Fault Tolerance
PCB	Printed Circuit Board
PKC	Public key cryptography

PoS	Proof of Stake
PoW	Proof of Work
PQC	Post-Quantum Cryptographic
RAM	Static Random Access Memory
RL	Reinforcement Learning
RTOS	Real-Time Operating System
SaaS	Software as a Service
SKC	Symmetric Key Cryptography
SLP	Shared Ledger Process
SME	Small and Medium Enterprises
TCP/IP	Transmission Control Protocol/Internet Protocol
TD	Thing Descriptions
UDP	User Datagram Protocol
ULP	Ultra Low Power
UUID	Universally Unique Identifier
VR	Virtual Reality
WEEE	Waste Electrical and Electronic Equipment
Wi-Fi	Wireless Fidelity
WoT	Web of Things

List of Equations

Chapter One: Introduction

The biologically inspired lightweight polymorphic security system for IoT devices represents an interdisciplinary approach, merging the domains of embedded systems engineering and immunological principles from biosciences. This innovative framework introduces a novel capability in IoT security systems, enabling them to emulate biological processes that detect, reject, and retain the memory of foreign agents. The paragon of this framework lies in its adept translation of immune mechanisms into embedded programming logic, leveraging adaptive complexities that dynamically adjust according to hardware configurations.

The Internet of Things (IoT), which enables a wide variety of embedded devices, sensors and actuators (known as smart things) to interconnect and exchange data, is a promising network scenario for bridging the physical devices and virtual objects in the cyber world. Such smart devices and sensitive data are vulnerable to security threats. Security is, therefore, the central area of focus for researchers in the field of IoT. Consequently, it is essential to develop cryptography technologies to secure the data from unauthorised access. Moreover, this can be achieved by transforming the data into an unrecognisable and unrelatable form. It is not easy to find one straightforward approach that will fit all IoT applications. There are various types of devices connected to an IoT network. Some devices can afford heavyweight and high-security methodologies, but most IoT devices are resource-constrained. They need a security solution that acts fast. Simultaneously, it needs to be simple in its complexity and versatile. Last but not least, the most critical factor is trusted security. In general, the dedicated cryptographic algorithms need to be lightweight in terms of area, memory footprint, power and energy consumption. Therefore, the proposed research is pertinent as security and privacy in the IoT are not fully addressed.

Polymorphism is a very uncommon term and a rarely researched topic within the traditional tech, cybersecurity and engineering industries. Among the computer sciences, the nature of polymorphic code can change the initial code, but the code’s function produces the same result; for example, 15+2 and 13+4 both have the same result, 17.

The changing form and mutation are more familiar with bypassing security systems in polymorphic viruses (e.g. malware) than in the security systems[1]. However, this mutating form is well studied and documented in the human sciences, particularly biology. The notion of this lightweight polymorphic security system closely follows and resembles the biological functions of the immune system. In particular, a human white blood cell, due to its ability to detect (intrusion detection), change (changing encryption keys and passwords) and fight (eject from the network) foreign cells inside the human body[2].

1.1 The Driving Force

Rapid technological development and deployment invariably lead to the emergence of new vulnerabilities, which malicious actors and unauthorised users may exploit. The Internet of Things (IoT) has diverse applications across various industries, from medical to commercial, industrial to automotive, and the consumer industries. According to the Information Handling Services (IHS), there were more than “27 billion connected IoT devices in 2017” [3]. Therefore, security is ever more paramount, especially as the industry grows exponentially due to big data, data analytics, and understanding consumer trends for effective marketing. As the IoT’s demand grows, the value for these industries is the raw data itself, hence protecting the data from foreign and unauthorised agents.

The idea of security systems for IoT is nothing new. Still, it does propose a few limitations compared to traditional security architectures as the developers are severely limited to the processing power within these devices. Consequently, the need for lightweight ciphers to encrypt communication is necessary. This research aims to encrypt the IoT systems’ communication channels and automatically change the encryption cipher key and the password required to connect a client to a server, hence making it polymorphic.

Industry and research are interconnected and coupled, two sides of the same coin. That without research, the industry will become stagnant and unable to innovate and expand, whereas, on the other side of the coin, it does not have the incentive to put forth solutions for today’s problems without industry research.

1.2 Theory of Operations

Before any terms of aims and objectives are drawn, there needs to be a clear understanding of how the lightweight polymorphic system would ideally work. The design and idea of the polymorphic security system, like most engineering solutions for world problems, takes inspiration from the most aged engineer who is nature herself—in particular, looking at how the immune system (adaptive immune responses and innate immune responses[4][5]) in the human body. How the immune system fights foreign agents (e.g. viruses and diseases) can be modified (updated for lack of a better term) to combat new viruses and infections through vaccines.

This system’s fundamental nature is on a server and client model, whereby the server and client communicate in an encrypted manner. The encryption key is in the ledger (similar to blockchain technologies) and shared with authorised clients. Authorised clients are approved depending on various factors, such as checking if it's on the block list, having the correct credentials, and the correct UUID. As illustrated in *Figure 1.1*, the system's top-down architecture comprises the overall system's approach.

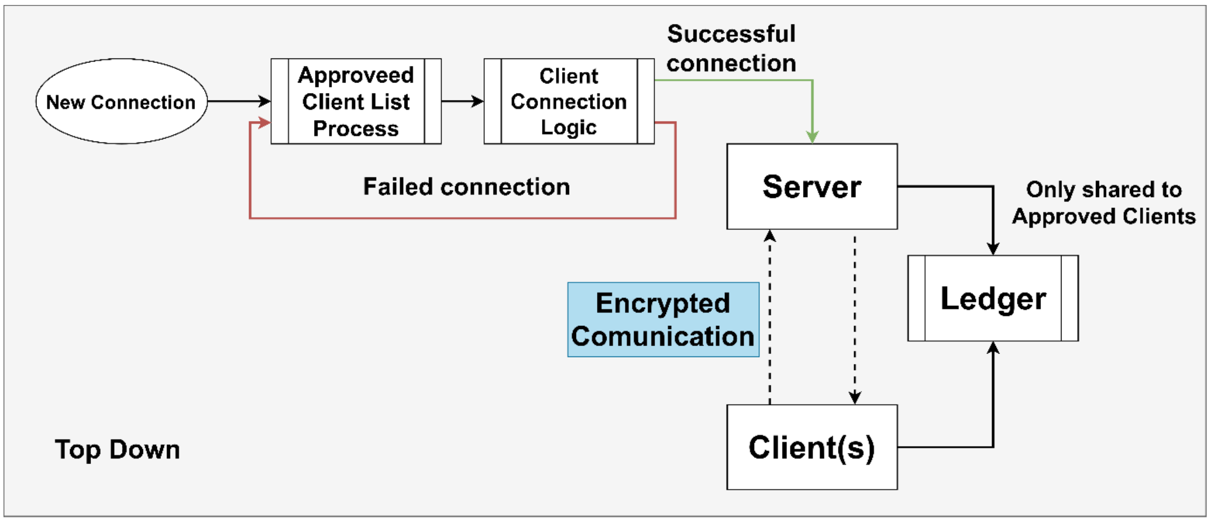


Figure 1.1: Top-Down Overview

Within the Approved Client List Process, flags can be raised if only authorised clients can connect to the server. The Wi-Fi details automatically change if the server detects any foreign clients trying to connect. Please refer to *Figure 1.2*, which shows how clients are approved.

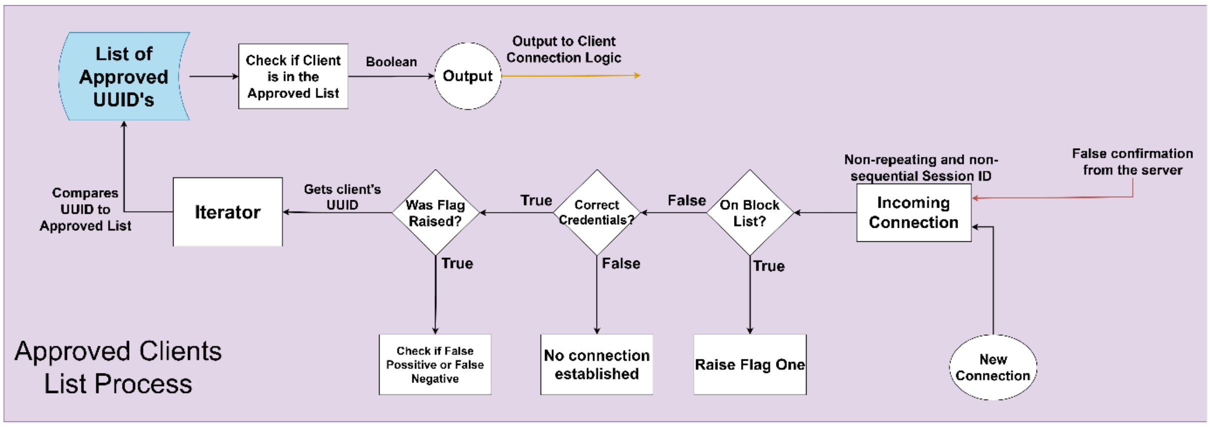

List of Approved UUID's
Check if Client is in the Approved List
Boolean
Output
Output to Client Connection Logic
Compares UUID to Approved List
Iterator
Gets client's UUID
Was Flag Raised?
True
Correct Credentials?
False
On Block List?
Non-repeating and non-sequential Session ID
Incoming Connection
False confirmation from the server
True
False
True
Check if False Possitive or False Negative
No connection established
Raise Flag One
New Connection
Approved Clients List Process

Figure 1.2: The process for how clients are approved

All authorised clients have the same ledger. This ledger contains two variables as objects, which are inside an array. The first object is the Wi-Fi details (password), and the second object includes the encryption key to which the client and server communicate. Please refer to *Figure 1.3* to visually show the ledger's process.

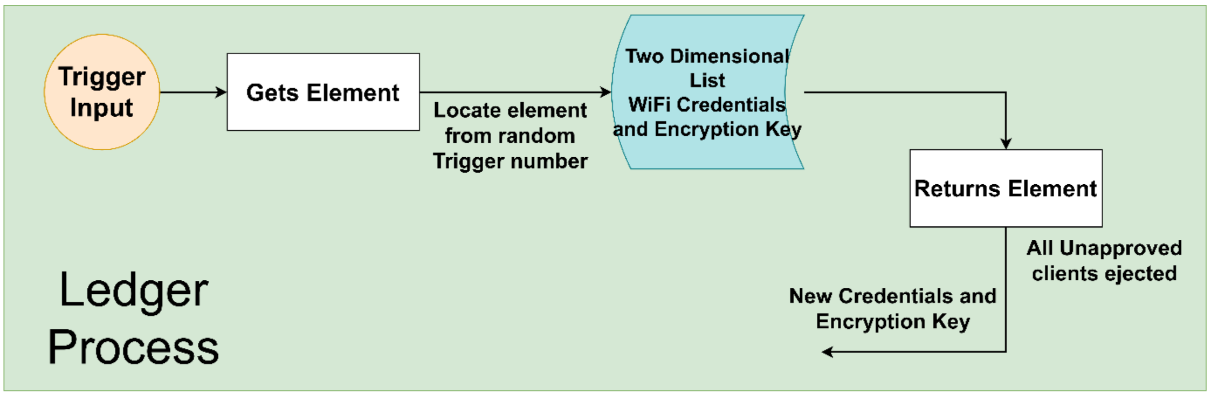

Trigger Input
Gets Element
Locate element from random Trigger number
Two Dimensional List WiFi Credentials and Encryption Key
Returns Element
All Unapproved clients ejected
New Credentials and Encryption Key
Ledger Process

Figure 1.3: Ledger Process

When any form of intrusion is detected, for example, an unauthorised client successfully connects to the server, a signal (trigger) is sent between the server and client to successfully apply the change between one polymorphic form to another (A -> B).

The signal (trigger) is a message which contains a number (randomly chosen) that corresponds to an element in the array (ledger) that has the new Wi-Fi details and Encryption Key; this will be its new polymorphic form. Please refer to *Figure 1.4*, which shows the server's logic when a client connects.

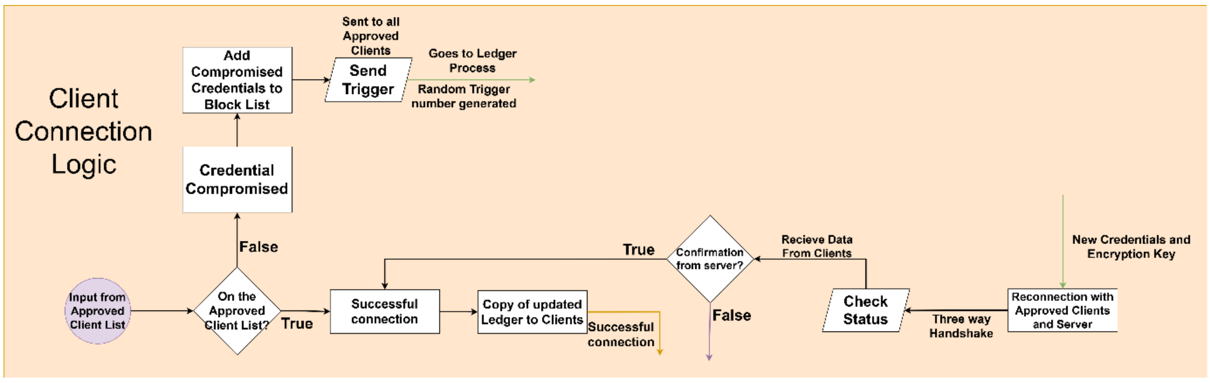

Client Connection Logic
Sent to all Approved Clients
Add Compromised Credentials to Block List
Send Trigger
Goes to Ledger Process
Random Trigger number generated
Credential Compromised
False
Input from Approved Client List
On the Approved Client List?
True
Successful connection
Copy of updated Ledger to Clients
Successful connection
True
Confirmation from server?
False
Recieve Data From Clients
Check Status
Three way Handshake
Reconnection with Approved Clients and Server
New Credentials and Encryption Key

Figure 1.4: Client Connection Logic

There are, however, some flaws in this system; the first problem is the client's, which might fall out of sync and no longer have the proper signal (trigger); therefore, it cannot connect to the server. The second problem is with New clients who are not on the approved client list and those associated with clients who are no longer in sync, which reverts to the first problem. Problem one is solved by having time-limited Wi-Fi details, which, once joined, give the current signal to bring the client back into sync.

To solve problem two, the server administrator can only accomplish this by updating the server's approved client list by physically inputting it into the server. Once added, the client needs to be synced, which resolves the problem.

In addition to addressing security challenges through polymorphic mechanisms, the proposed system also recognises the critical need for energy efficiency in resource-constrained IoT devices. Traditional IoT security frameworks often overlook the significant impact of encryption processes on battery consumption, particularly when devices are deployed in remote or inaccessible environments where energy availability is limited. To overcome this limitation, the framework integrates a novel Adaptive Amoeba Battery Curve Mapping Management System (AABCMS), inspired by the adaptive behaviours of biological organisms under energy stress. The AABCMS dynamically monitors real-time battery performance and system load, adjusting device operational modes, including sleep states and encryption method selection, to optimise energy usage without compromising security. This integration of energy-aware decision-making within the security architecture represents a fundamental departure from conventional static approaches and directly addresses the dual challenge of security and sustainability in IoT systems.

1.3 Aims and Objectives

The aim of this research is to design, implement, and rigorously evaluate a novel, bio-inspired, lightweight polymorphic security system specifically tailored for Internet of Things (IoT) environments. The system is grounded in principles from immunology and adaptive energy management. It is engineered to provide resilient, energy-aware protection against evolving cybersecurity threats, while remaining feasible for real-world deployment on power-constrained embedded devices. The aims and objectives of this research closely follow the system overview as the bio-inspired lightweight polymorphic security system for IoT needs to achieve:

- To architect and implement a modular client-server security framework for IoT devices, utilising polymorphic encryption strategies that adapt in real time to resource constraints.
- To model and simulate biologically inspired security responses, including detection, rejection, and memory, based on principles from immunology.
- To develop a synchronised, lightweight shared ledger mechanism for dynamic credential management across networked devices.
- To design and integrate the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS) for real-time optimisation of device sleep modes, encryption selection, and battery health monitoring.
- To quantitatively benchmark and compare multiple lightweight encryption schemes on both high-power and low-power embedded platforms.
- To validate the robustness and resilience of the system through scenario-based experimental testing, including simulations of desynchronisation, session expiration, low-battery operation, and security breach responses.

1.4 Research Contribution For The Engineering Community

There are several research contributions to the engineering community regarding IoT security, Intrusion Detection, Polymorphic Engines, viruses and worms. However, there is a gap, a sort of goldilocks zone, whereby a lightweight system for an IoT network can act in a polymorphic way to change and adapt accordingly when an unauthorised client connects. Nonetheless, there are some suitable papers from which this research can definitely avail and learn.

“Polymorphic Algorithm of JavaScript Code Protection”[6] focuses on web-based security for the shared dynamic web, developing a language called JavaScript. The JavaScript code does not compile as they have to be executed by the web browsers even if it is uncompiled. Therefore, it results in the source code presented into bytes or binary codes, making data extraction from JavaScript somewhat vulnerable without additional encryption. This study develops an algorithm using the polymorphic viruses’ reference design to make random encryption for Web page encryption and a Web-based information security system. The study’s strengths focus on defending against polymorphic attacks using a reference design system to make random encryptions on a JavaScript Web-based system. This detection system allows the security system not to always be running in the background and only employs encryption when necessary, therefore, saving resources of the web server until it is necessary. However, weaknesses do arise; this is not a lightweight system, as the web server (even though they have intensive processing power) must do live encryption. This research paper will incorporate this research into the responsive encryption that only reacts when it detects suspicious flags.

“Design of the late-model key exchange algorithm based on the polymorphic cipher” [7] proposes a new polymorphic cipher method created by C.B. Roellgen in 2004 opened up a whole new area for the study of irregular symmetrical cryptography theory as an asymmetrical cipher algorithm. The polymorphic cipher is characteristic of randomness; the newly proposed method uses a Pseudo-random Number Generator to construct the polymorphic virtual S-box. The polymorphic cipher design’s purpose makes the session keys immune to attacks. This study’s strength is an efficient polymorphic key exchange algorithm based on the session key exchange protocol. However, similarly to the previous research is resource-intensive with the Pseudo-random Number Generator, and the design cost of the agreement is high. The author concentrated on low powered systems with a lightweight polymetric algorithm for this proposed research.

“Webpage Encryption Based on Polymorphic JavaScript Algorithm.” [8] Looks into protecting HTML code encryption based on polymorphic JavaScript programs, which can transmute and defend themselves like polymorphic viruses. The study’s critical point is that both encrypted HTML codes and polymorphic JavaScript programs are difficult to be cracked. Therefore, the webpage content is protected. This study's strength is that the nature of HTML encryption uses compression, permutation, and check digits to enhance the security effect. However, the system can crack by using a JavaScript debugger to show the memory locations of when and where the Polymorphic JavaScript Algorithm is activated. In this proposed research, IoT based and not a web-based system; therefore, software debugging would be challenging to reverse engineer, especially using a logic analyser.

Overall, the proposed research will add to the contributed knowledge by developing a lightweight polymorphic system on an IoT device that can detect when a new client connects to the server for authorisation from a predetermined database that authorises clients. Once the server has identified the unauthorised client, it will reset the shared SSID password and share the randomly chosen new password to the trusted authorised clients. Additionally, communication between the server and authorised clients is encrypted. Therefore, if the unauthorised client manages to collect some communication data, it will not be compromised or used to reverse the system.

The latter chapters of this research paper will pay closer attention to a broad spectrum of research papers. To cover a considerable breadth of related topics, analysed to see what has previously been accomplished, what is adaptable to this research and what this research paper can contribute and "*boldly go where no* [person] *has gone before*".

1.5 Overview of The Thesis

This thesis is systematically structured to present the development, implementation, and evaluation of the Bio-Inspired Lightweight Polymorphic Security System for IoT Devices. The seven main chapters, along with comprehensive appendices, progressively guide the reader through the research problem, conceptual foundation, technical design, and experimental validation of the proposed system.

Chapter One introduces the thesis, presenting the motivation behind the research, the theoretical underpinnings inspired by biological immunology, and the technical rationale for a polymorphic and lightweight security system. It outlines the core aims and objectives, as well as the broader significance of the research to the engineering community, particularly within the contexts of embedded systems and cybersecurity.

Chapter Two provides an extensive literature review, covering the historical development and state-of-the-art research in the Internet of Things (IoT), network security, cryptography, and polymorphism. This chapter also introduces battery management strategies for IoT and evaluates related studies in these areas. It lays the theoretical groundwork by identifying existing gaps in integrating security with adaptive energy management, a key motivation for the development of this thesis.

Chapter Three offers the technical and theoretical background necessary to contextualise the research. It surveys IoT hardware platforms, the evolution of cloud computing, the fundamentals of cryptography, the biological mechanisms of immunity, and blockchain concepts and introduces the conceptual linkage between brainwave modulation, energy adaptation and bio-inspired communication networks such as mycelium systems, establishing the interdisciplinary basis for the proposed system.

Chapter Four outlines the core framework of the proposed system, detailing the Approved Clients List Process (ACLP), Client Connection Logic (CCL), Ledger Process (LP), Adaptive Amoeba Complexity (AAC), and the novel Adaptive Amoeba Battery Curve Mapping Management System (AABCMS). This chapter discusses the bio-inspired parallels and system logic in detail, supported by high-level architecture diagrams.

Chapter Five transitions from conceptual design to practical implementation. It details the full deployment of the system on a custom ESP32-S3 development board, and includes the construction of client-server architecture, implementation of multiple lightweight and block cipher encryption methods, secure communication logic, session handling, and the integration of AABCMS. Code snippets and mathematical models are used to demonstrate key system functions.

Chapter Six presents the testing and refinement process. It evaluates encryption efficiency through cycle and power analysis on both ESP32-S3 and ATMEGA328P platforms. It further assesses the auto-detection, auto-ejection, and trigger mechanisms by simulating various attack and synchronisation scenarios. The full system is tested across multiple scenarios, including session handling during low-power states and desynchronisation recovery via honeypot mechanisms. Detailed results validate the system's responsiveness, security, and power optimisation capabilities.

Chapter Seven concludes the thesis by synthesising the key findings, reflecting on the personal and technical journey of the research, and proposing directions for future work. It explores how the system may be extended through AI integration, post-quantum cryptography, decentralised ledgers, and biologically-inspired models for next-generation cyber-physical security.

Finally, the Appendices provide detailed schematics of the hardware, complete implementation code, test data, and supplementary tables and diagrams. These materials support the reproducibility, transparency, and practical applicability of the research.

2.0 Chapter Two: Literature Review

Chapter two will cover various topics that encompass this research. It spans from the Internet of Things (IoT) devices, Security Systems within networks and the ideology of polymorphism from the aspect of a changing system. To further understand how each topic and its subtopics are related to one another and how they could collaborate. Understanding each topic's history and how other researchers have contributed to the overall industry would further develop knowledge for this research.

2.1 Topic History and Development

2.1.1 Internet of Things

2.1.1.1 Brief History of the Internet of Things

IoT (Internet of Things) is a term coined in 1999 by Kevin Ashton, who is a British technology pioneer who co-founded the Auto-ID Centre at the Massachusetts Institute of Technology (MIT)[9]. A general explanation for IoT, according to McKinsey, is that IoT is "sensors and actuators embedded in physical objects, from roadways to pacemakers, are linked through wired and wireless networks, often using the same Internet Protocol (IP) that connects the Internet."[10].

However, due to IoT being a relatively new industry, there is no universally comprehensive definition as various industries adopt IoT for their unique needs. Therefore, they may label IoT as something else. For example, Intel originally called it the Embedded Internet, whereas Cisco preferred the term Internet of Everything (IoE). Various industries also use other terms, as they all have slightly different meanings, such as:

- Industry 4.0
- Pervasive Computing
- Smart Systems
- Intelligent Systems
- M2M (Machine-to-Machine) Communication
- Industrial Internet of Things (IIoT)
- Web of Things (WoT)

Industry 4.0, also known as the Fourth Industrial Revolution, emphasises the industrial practices of implementing technology through automation, as well as its real-world applications, including Virtual Reality (VR), Augmented Reality (AR), Mixed Reality (MR), and 3D printing. This approach and the term itself originated in 2011 from the German government to incorporate a high-technology strategy that implements the computerisation of manufacturing[11].

Pervasive computing is the precursor to IoT, as it establishes principles for a connected world, primarily targeting the telecommunications industry for mobile communications. Nonetheless, Pervasive computing fundamentals follow IoT methodologies such as decentralisation, diversification, connectivity, and simplicity[12].

Both Smart Systems and Intelligent Systems are intertwined as they focus on how systems interact with the physical human-facing side that has dynamic physical and social environments. For example, assistive robotics, medical care, education, entertainment, visual surveillance, and biometric human identification.

Machine-to-Machine (M2M) communication allows communication between other devices through a network system, either wired or wireless, via an IP network system. M2M sets the foundation for Industrial Internet of Things (IoT) and Industry 4.0.

Lastly, the Web of Things (WoT) represents a conceptual and architectural evolution of the IoT by extending its integration directly into the web ecosystem. While IoT is primarily concerned with connecting devices through embedded sensors and network protocols, the Web of Things focuses on standardising the interaction with these devices using established web protocols such as HTTP, WebSockets, and RESTful APIs. This web-centric approach aims to simplify the interoperability of heterogeneous systems by making “things” accessible and manageable through uniform web interfaces.

Technically, WoT introduces the concept of Thing Descriptions (TDs), which serve as metadata models that describe the capabilities, properties, and communication endpoints of IoT devices. These TDs are typically written in JSON-LD and aligned with semantic web technologies, allowing machines to interpret and reason about device behaviour automatically. As a result, WoT supports not only human-centric interactions but also machine-to-machine interactions within a web-based framework.

For example, a smart thermostat in a WoT environment may expose its temperature sensor, operating range, and control functions via a RESTful interface. These capabilities can then be accessed from any compatible application or device using standard web protocols, eliminating the need for proprietary software stacks or middleware.

In summary, the Web of Things builds upon the Internet of Things by leveraging the web as a unifying medium for connecting, describing, and interacting with smart devices. Its emphasis on open standards, semantic modelling, and resource-based architectures contributes significantly to the scalability and maintainability of large-scale IoT deployments.

2.1.1.2 Generic Life-Cycle for IoT Devices

Every product has a life cycle from the initial development phase to its growth, maturity and decline. To understand how businesses are implementing IoT devices, first, there needs to be an understanding of the generic life cycle for IoT devices. There are five main stages (Please refer to *Figure 2.1 General Life Cycle for IoT Devices*) for the IoT product life cycle: (Re)Construction, Deployment, Growth, Manage, and Decommission/Recycle.

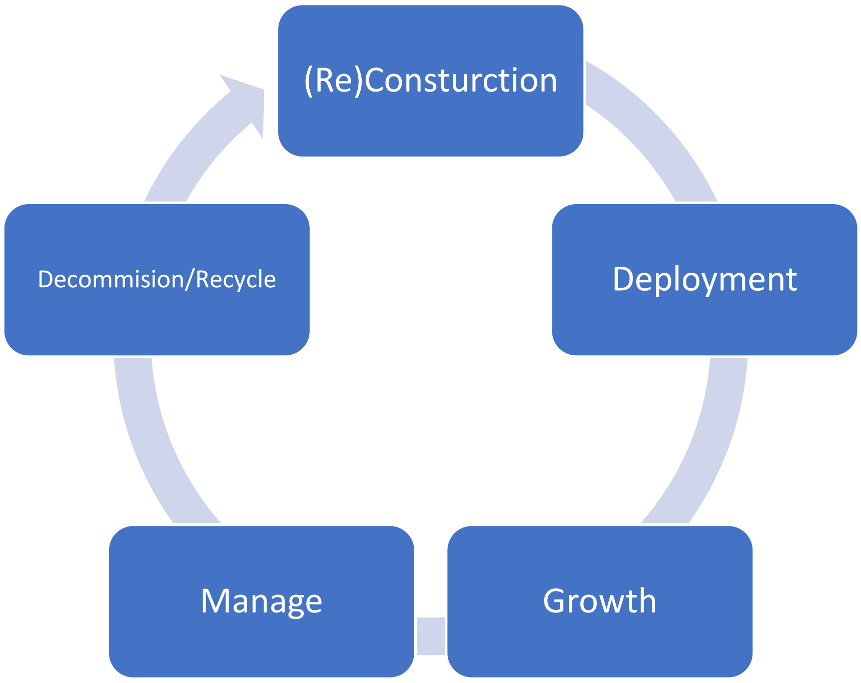


Figure 2.1: General Life Cycle for IoT Devices

The first stage (Re)Construction begins with constructing the device hardware and software, or reconstructing hardware components and updating software, like security and firmware[13]. This stage's development encompasses the specification, requirements, business needs, and constraints, as the product must consider the end-use and life cycle. Some of the things to consider are what type of data this device will gather, designing the appropriate circuit to allow external sensors, powering the device, and implementing and testing the product to determine if the specification is accomplished.

The second stage, Deployment, is a multipart process that requires the production/manufacturing and installation of the device. Due to the yearly cost reduction of manufacturing sensors, the ability to mass-produce these IoT devices is relatively cheap. In the first part of this stage, the devices are mass-produced. Each IoT device is given a unique ID or unique universal ID (UUID), and the initial software with an over-the-air (OTA) programming ability can also include certificates or encryption keys[14]. The second part is the physical installation of the IoT devices in their environment, such as buildings, vehicles, and equipment. The installation may include factors for powering the device, as if the device is in a building with electricity, and it can be connected directly to the electrical network. An IoT device is placed somewhere remote or mobile, like a weather station, which needs a mobile power source and debugging features with limited human intervention. Geolocation might be implemented to track the device and find the device; for example, if a weather station is implemented in the desert and an intense sand storm may have displaced the sensor or damaged it, human intervention is required to repair it.

The second stage directly influences the third stage, growth, which utilises the product by pairing the sensor with the control server to verify communication streams. After the pairing is successful, the device is registered as a connected device in the field, and data communication to the server begins. The importance of registering connected devices is to keep track of what devices are active and ensure legitimacy.

The fourth stage, Manage, is about monitoring, maintaining, and updating the IoT devices. The central server that enables remote monitoring of connected devices includes operating status, battery level, configuration settings, and software version. Additionally, the configuration of the alerts determines whether the IoT product is working as expected or if any suspicious activity occurs. For example, suppose the IoT device is a part of a BotNet. In that case, it will transmit an enormous amount of data, which can raise an alert to the central server to reset the device to its factory settings, therefore, no longer infected with the malware. Another common type of alert is examining the transmitted data to see if it is erroneous. One of the advantages of IoT devices is their ability to be mobile. However, this poses a challenge for updating, as the manufacturer cannot recall all the products to update. The solution is to have over-the-air (OTA) updates to allow the connected device to be remotely updated, configured, and recalibrated. OTA is also the primary solution for maintenance operations if the device can be repaired with software; otherwise, physical activity is considered.

The fifth and last stage of the cycle is decommissioning/recycling. At the end of the IoT device life cycle, the need for significant upgrades to more recent hardware, or the end of gathering a particular dataset, is apparent. The secure removal of the connected device from its platform for decommissioning is set forth. In addition to decommissioning the connection and permissions between IoT devices and servers, they are revoked through securely unpairing the IoT device[14]. Decommissioning is achieved through a final software update. The IoT device is physically removed from the environment to protect the environment by recycling, and to prevent data loss and data theft. Since 2018 (an updated version of the 2013 regulation of the same name), the regulation for waste electrical and electronic equipment (WEEE) requires manufacturers to conduct or outsource electronic recycling and waste recovery in a safe and environmentally responsible manner. However, for recycling IoT systems (if not obsolete), most devices and sensors are redesigned to create new models, thereby saving future project costs.

2.1.1.3 The Present to the Future

Over the past decade, IoT progression has been exponential as sensors' manufacturing costs have decreased. Therefore, incorporating IoT sensors for projects is limited to large enterprises and SME's (Small and Medium Enterprises). According to the "2019 manufacturing trends report" by Microsoft [15]. The average price in 2005 was $1.30, compared to 2020, the cost was $0.38, which is a 49.4% decrease in price. Please see *Figure 2.2* for a cost trends graph from 2004 to 2020.

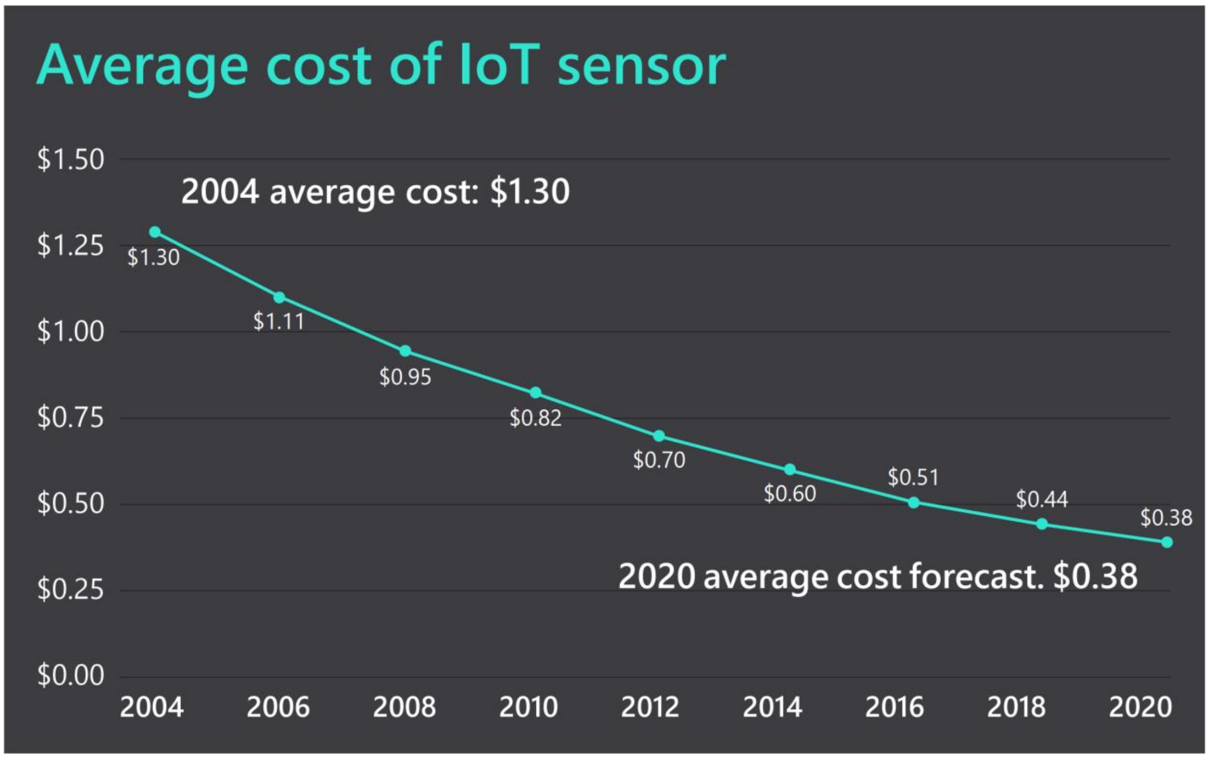


Figure 2.2: Average cost of IoT sensors - Taken from the 2019 Manufacturing Trends Report by Microsoft

Due to IoT devices' cost-effectiveness, the range of applications is only limited by the product's design specifications. Thus, the accelerated rate at which the digital and physical are becoming ever more interconnected. For example, to name a couple of types of IoT applications would be:

- Applications within commercial IoT, as the Transport industry uses monitoring systems for trains and vehicle communication to avoid traffic in smart and electronic cars. Additionally, the Healthcare industry, using Pacemakers and automated insulin pumps.
- Consumer IoT is more apparent as it includes home applications such as a smart fridge, personal assistance like Alexa or Google Home. Wearable technologies communicate with a smartphone via an application, such as a smartwatch.

- The sector of military, government or intelligence agencies could use IoT through surveillance and facial recognition to track and digitally follow suspects. Implementation of biometrics for combat training to further understand how a subject reacts to a specific scenario.
- Smart agriculture, control systems, and industrial big data for statistical evaluation all come under the industrial IoT. Especially within the agricultural industries, there is a need to expand to feed the world's growing population without taking too much space and having an ecological footprint. Due to this demand, more inner-city farms (or vertical farms) are operating closer to the seller and growing plants in a more closed and controlled environment, therefore, resulting in saving costs for pesticides, transport and storage.
- Lastly, the notion of smart cities, which comes under infrastructure IoT, enables an ecosystem for interconnected devices and services.

The question that arises from the low-cost adaptability and the highly adaptable implementations of IoT devices is How many IoT devices are there? What are users spending on IoT devices? Which industries are adopting IoT?

Table 1 shows the number of installed IoT devices from 2018 to 2020, then an estimate for 2021 and 2025.

Table 1: IoT Devices Installed Worldwide[16]

Year	Number of Devices Worldwide (in Billions)
2018	7
2019	26.66
2020	31
2021	35*
2025	>75*

Figure 2.3 shows a forecast of end-user spending on IoT solutions worldwide from 2017 to 2025. The IoT industry “reached 100 billion dollars in market revenue for the first time in 2017, and forecasts suggest that this figure will grow to around 1.6 trillion by 2025.”[17]

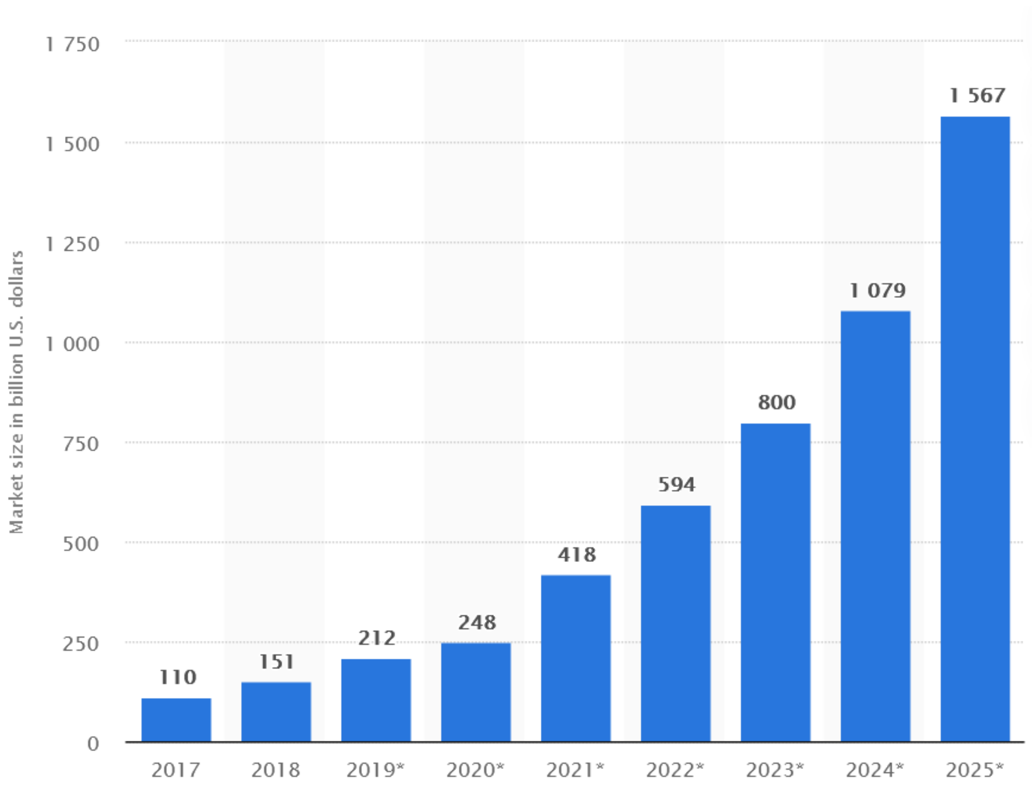


Figure 2.3: Global IoT end-user spending worldwide 2017-2025 [17]

2.1.2 Network Security Systems through Encryption and Cryptography

2.1.2.1 Basic Network Systems

Before delving into the security side of networking, encryption, and cryptography, one needs to understand how network systems are structured and organised. The structure layers are the basis for the Open Systems Interconnection (OSI) and Transmission Control Protocol/Internet Protocol (TCP/IP) models.

This chapter will briefly discuss the differences and similarities between the OSI and TCP/IP reference models and then discuss each of the layers in more detail; please see *Figure 2.4*, which shows a basic outline of the layers between the OSI and TCP/IP models.

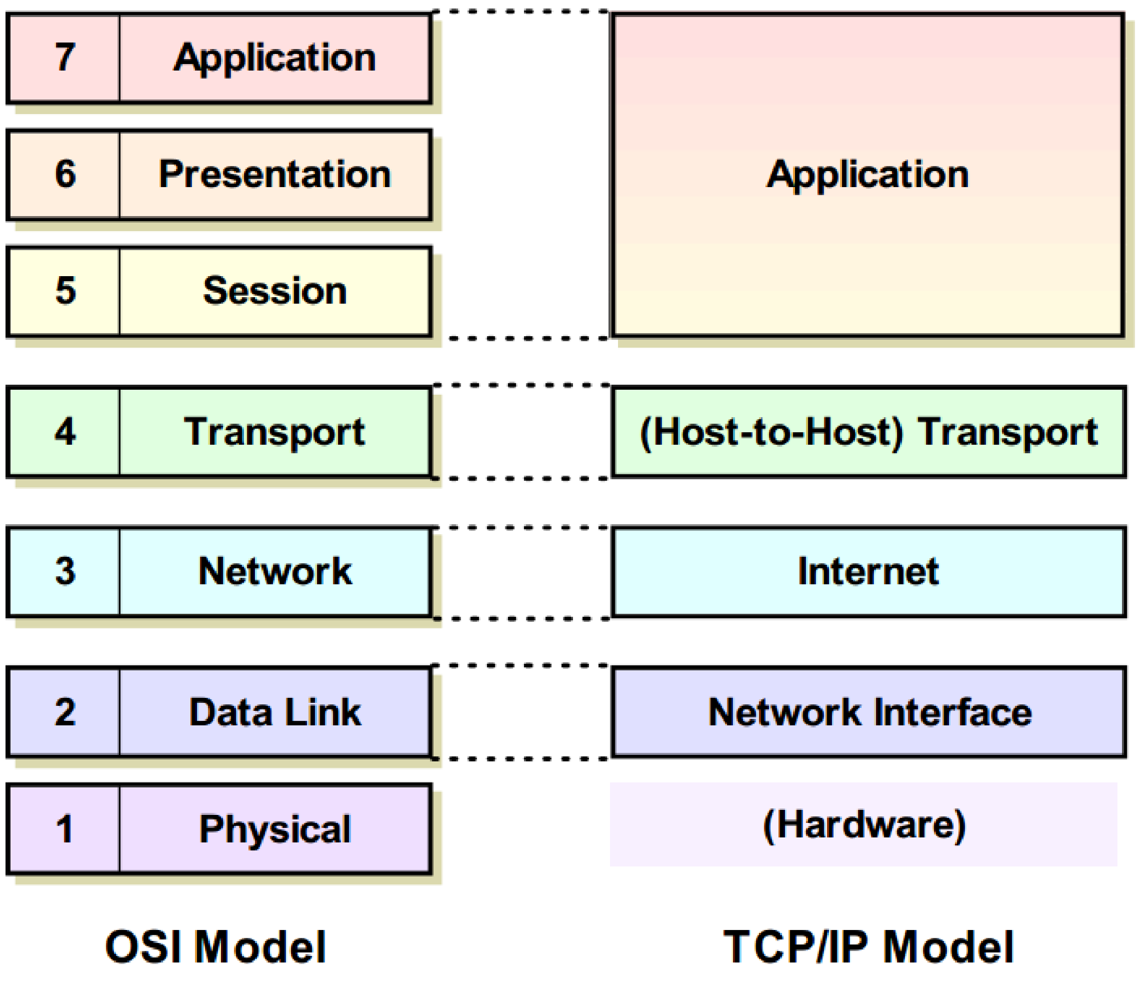


Figure 2.4: Layers of the OSI and TCP/IP models - Taken from [Charles M. Kozierok]

There are a few differences between the OSI and TCP/IP reference models; for example, the OSI model has seven individual layers (vertical approach) at its face value. In contrast, the TCP/IP has four (horizontal); this is mainly because the TCP/IP model is a protocol-oriented standard. For example, the Hypertext Transfer Protocol (HTTP) is in the application layer, and Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are in the transport layer. The OSI model is more logical and conceptually based on each layer's functionalities. The OSI model distinguishes three concepts: protocols, interfaces, and services that help standardise motherboards, routers, switches, and other hardware. In contrast, TCP/IP helps establish a connection between different types of computers.

However, the similarities of the two are that both reference models divide data into packets, and each packet may take a unique route from the source to the end-user destination. Another similarity is that the layers are compatible with each other. The physical layer and the data link layer of the OSI model and the TCP/IP model correspond. The same goes for both the network and transport layers.

The differences and similarities between the layers have been discussed previously, but what are their functionalities?

Starting from the bottom, the physical layer has the critical responsibilities of signalling and encoding. Therefore, physically transmitting the data through electrical or light signals sent between local devices, as the data type handled by this layer is in bits. The physical layer is where the hardware specifications, topology and design of the entire network begin.

The data-link layer focuses on low-level data messages between local devices; these local devices are directly connected nodes that perform node-to-node data transfer, where the data is packaged into frames. The Data-link layer has the responsibilities of media access control (MAC), data framing and handling the frames, logical link control (LLC), error detection, handling, addressing, and defining the previous physical link layer requirements.

The Network layer handles data types of datagrams and packets; it is responsible for receiving frames from the previous data link layer. The Network layer's primary responsibilities are logical addressing, routing, datagram encapsulation, fragmentation and reassembly, error handling, and diagnostics. The network layer's scope is the messages between local or remote devices.

Managing the delivery and the error checking of the data packets is the main factor of the transport layer. The data type handles datagrams and segments in the transport layer, as it functions as a multiplexer and demultiplexer, providing process-level addressing, connections, segmentation and reassembly, acknowledgements and retransmissions, and flow control. The transport layer's scope is the communication between software processes, such as TCP and UDP.

The Session layer is responsible for session establishment, management and termination, as it controls the communications between different computers and machines. The data type that the session layer handles is sessions, as the scope of this layer focuses on sessions between local or remote devices, such as sockets or NetBIOS.

The presentation layer (sometimes also called the syntax layer) is responsible for data translation, compression, and encryption. The data type handled in this layer is encoded user data in the scope of application data representations. A few examples of standard protocols and technologies in this layer are SSL and MIME.

Lastly, there is the application layer, whereby the end-user interacts directly with the software application, and it handles the user data and application data. The standard protocols and technologies used in this layer are DNS, SNMP, HTTP, Telnet, FTP, DHCP, and more.

2.1.2.2 Network Security with Encryption and Cryptography

The fundamental nature of encryption derives from the need to communicate and send messages back and forth securely. As technology is a tree that branches new inventions to birth instead of miraculously spawning into existence, the history of encryption comes from linguistics. Linguistics is "*the scientific study of languages, how they evolve and are structured*" [18].

The simplest form of encryption in the linguistic field is when two individuals in a group of people have a private conversation by whispering, as this form of communication is secure to a small area of effect of hearing.

Another example is when two people speak another language that is not the commonly spoken tongue for that country, te respicere post tergum? Most readers of this research paper may not know the sentence; hence, it is secure only for those who know, although due to the existence of online translators, deciphering the unknown language is not a challenge.

Throughout history, encryption has evolved; the aim has remained the same: to communicate with the intended recipient securely. However, the methodology changed and became more complicated due to the volume of data that needed to be secured faster; this demand led to the study of cryptography.

"*Cryptography is the science* [art] *of keeping secrets*" [19], especially in telecommunications, is necessary when communicating over any unsecured devices and platforms; this includes any networks, including the internet, as the internet is a network of networks. There are five main functions cryptography has to offer[9]:

- Privacy and confidentiality: Ensuring that no one can read the message except the intended receiver. Therefore, information that only the intended person can access.
- Authentication identifies who the sender and receiver are, as well as the origin and destination of the information.
- Integrity is the method of assuring the receiver that the received message has not been altered in any way from the original.
- The implementation of non-repudiation to prove that the sender was the originator of the sent message.
- Key exchange is the method by which crypto keys are shared between the receiver and the sender.

Before cryptography, all data started with unencrypted data, which refers to plaintext. The plaintext is encrypted into ciphertext, which will, in turn, be decrypted back into usable plaintext using a key.

The encryption and decryption are based upon the type of cryptography scheme employed and some form of a key, this process is written as:

Equation 2.1: General Equation for Encryption and Decryption

$$C = Ek(P)$$

$$P = Dk(C)$$

P = plaintext, C = ciphertext, E = the encryption method, D = the decryption method, and k = the key [9]

There are three types of algorithms for cryptography:

- Symmetric Key Cryptography or Secret Key Cryptography: Uses a single cryptographic key for both encryption of plaintext and decryption of ciphertext.
- Asymmetric Key Cryptography or Public Key Cryptography (PKC): Uses one key for encryption and another for decryption.
- Hash Functions or Hash Key Cryptography: Has no key within the algorithm but uses a fixed-length calculated based on the plain text.

2.1.2.3 Symmetric Key Cryptography (SKC)

Symmetric Key Cryptography (SKC) uses duplicate cryptographic keys for both encryption of plaintext and decryption of ciphertext. Once the encrypted text is received, the receiver applies the same key to decrypt the message and recover the original plaintext.

With this cryptography, both the sender and the receiver must have the same key: the secret. The most considerable difficulty with this approach is the distribution of the key. Symmetric key cryptography schemes are categorised into either stream ciphers or block ciphers.

Stream ciphers are a type of symmetric-key cipher that continually converts a byte of the plain text to encrypted data. Stream ciphers come in several variations; self-synchronising stream ciphers calculate each bit in the keystream as a function of the previous n bits. It is also called self-synchronising because the decryption process can stay synchronised with the encryption process by knowing the length of the n-bit keystream.

One problem is error propagation; a misplaced/unsynchronised bit in transmission will result in an absent bit at the receiving side, resulting in a separate message when decrypting the text. An essential feature of synchronous stream ciphers is that they assure only the confidentiality of data but not its integrity. An active attacker can flip the ciphertext’s bits, which converts the corresponding plaintext bits. To prevent active attacks, one needs a message authentication code (MAC).[20]

Synchronous stream ciphers generate the keystream independent of the message stream but use the same keystream generation function at sender and receiver. At the same time, stream ciphers do not propagate transmission errors. However, due to being periodic, the keystream will eventually repeat, and any errors will be overridden.

A block cipher encrypts one fixed-size block of data at a time. A block cipher is a given plaintext block that will always encrypt to the same ciphertext when using the same key, whereas the same plaintext will encrypt to the different ciphertext in a stream cipher.

Block ciphers can operate in several modes: Electronic Codebook, Cipher Block Chaining, Cipher Feedback, Output Feedback, and Control Mode.

The Electronic Codebook (ECB) mode uses the secret key to encrypt the plaintext block to form a ciphertext block. It is the simplest operation model as the plain text message is divided into 64-bit blocks, and each block is encrypted independently, but uses the same key for the encryption process. When transmitting a single bit error, ECB could result in a mistake for the entire decrypted plaintext. It is susceptible to brute force attacks, deletion and insertion attacks due to the lack of diffusion. The formula is as follows, and a flowchart is provided in *Figure 2.5*.

Equation 2.2: Electronic Codebook (ECB) Encryption Equation

$$C_j = CIPH_K(P_j)$$

Equation 2.3: Electronic Codebook (ECB) Decryption Equation

$$P_j = CIPH^{-1}{}_K(C_j)$$

C = ciphertext, *P* = plaintext, *$CIPH_K(X)$* = the forward cipher function, *$CIPH^{-1}{}_K(X)$* = Inverse cipher function, *K* = the secret key, and *j* is the sequence of the data from left to right.[21]

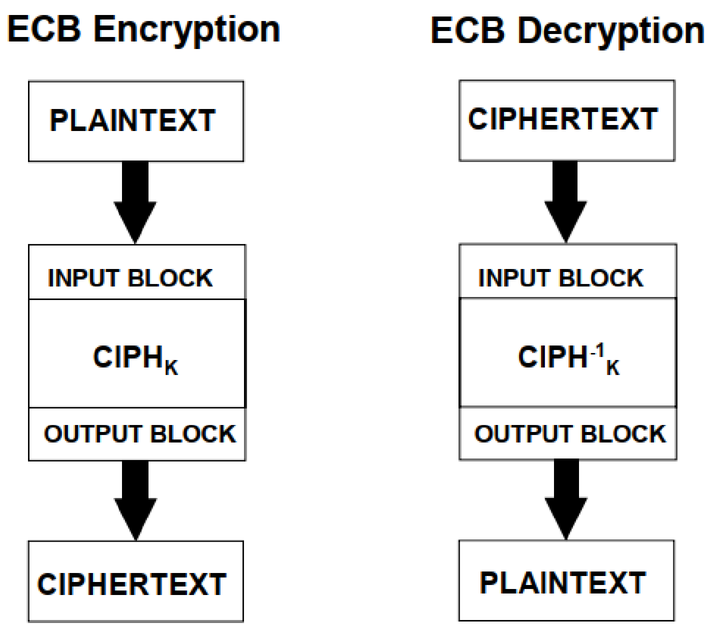


Figure 2.5: ECB encryption and decryption flow chart, taken from "Recommendations for Block Cipher Modes of Operation Methods and Techniques" [21]

Cipher Block Chaining (CBC) adds a feedback loop for cryptography. CBC is achieved by first encrypting the plaintext using an XOR cipher in the initialisation vector, then its encrypted again using a key, and this output is XOR also into another key and so on. This chaining destroys any sequential patterning and is protected against brute force, deletion and insertion attacks; however, a single bit error in the ciphertext will still cause the entire block to produce the wrong plaintext. The flowchart in *Figure 2.6* follows the formula for CBC.

Equation 2.4: Cipher Block Chaining (CBC) Encryption Equation

$$C_1 = CIPH_K(P_1 \oplus IV);$$
$$C_j = CIPH_K(P_j \oplus C_{j-1})$$

Equation 2.5: Cipher Block Chaining (CBC) Decryption Equation

$$P_1 = CIPH^{-1}{}_K(C_1) \oplus IV;$$
$$P_j = CIPH^{-1}{}_K(C_j) \oplus C_{j-1}$$

C = ciphertext, *P* = plaintext, *$CIPH_K(X)$* = the forward cipher function, *$CIPH^{-1}{}_K(X)$* = Inverse cipher function, *K* = the secret key, *j* is the second data of the sequence from left to right, *C_1* is the first data from the sequence, *$X \oplus Y$* bitwise exclusive-OR of two strings X and Y must be of the same length, *IV* is the Initialisation vector.[21]

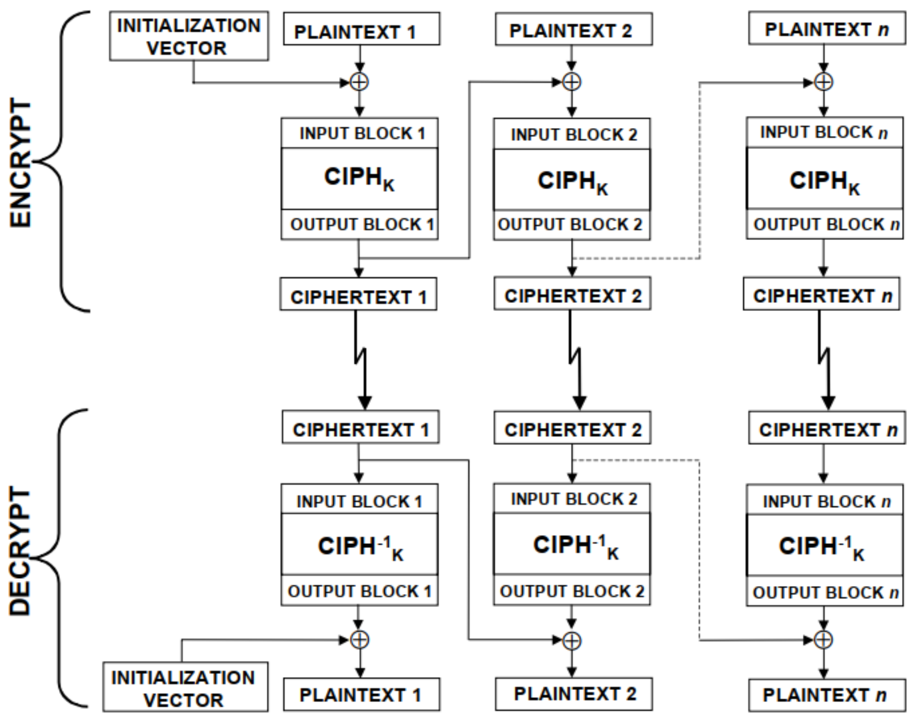


Figure 2.6: CBC encryption and decryption flow chart, taken from "Recommendations for Block Cipher Modes of Operation Methods and Techniques" [21]

Cipher Feedback (CFB) mode is a block cipher implementation as a self-synchronising stream cipher. This block cipher is very similar to CBC but performed in reverse. The reversal single bit error in the ciphertext affects both this block and the following one. Please refer to *Figure 2.7* for the flowchart, and the defined formula for CFB mode follows:

Equation 2.6: Cipher Feedback (CFB) Encryption Equation

$$I_1 = IV;$$

$$I_j = LSB_{b-s}(I_{j-1})|C^{\#}{}_{j-1}$$

$$O_j = CIPH_K(I_j)$$

$$C^{\#}{}_j = P^{\#}{}_j \oplus MSB_s(O_j)$$

Equation 2.7: Cipher Feedback (CFB) Decryption Equation

$$I_1 = IV;$$

$$I_j = LSB_{b-s}(I_{j-1})|C^{\#}{}_{j-1}$$

$$O_j = CIPH_K(I_j)$$

$$P^{\#}{}_j = C^{\#}{}_j \oplus MSB_s(O_j)$$

C = ciphertext, *P* = plaintext, $CIPH_K(X)$ = the forward cipher function, $CIPH^{-1}{}_K(X)$ = Inverse cipher function, *K* = the secret key, *j* is the second data of the sequence from left to right, C_1 is the first data from the sequence, $X \oplus Y$ bitwise exclusive-OR of two strings X and Y must be of the same length, *IV* is the Initialisation vector, *I* is the input block, *LSB* is the Least Significant Bit, *MSB* is the Most Significant Bit, $X|Y$ is the concatenation of X and Y, *b* is the block size in bits, *s* is the number of bits in a data segment, and *O* is the output, *#* is the segment of a block.[21]

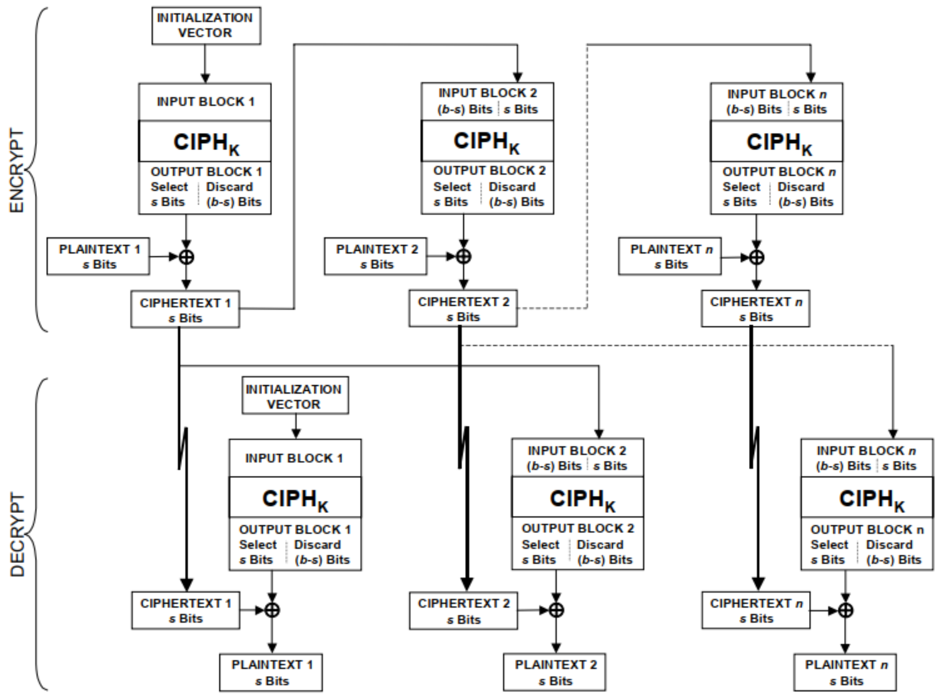


Figure 2.7: CFB encryption and decryption flow chart, taken from "Recommendations for Block Cipher Modes of Operation Methods and Techniques" [21]

Output Feedback (OFB) mode is a block cipher implementation conceptually like a synchronous stream cipher. OFB prevents the plaintext block from generating the same cipher text block by using a feedback system to generate the keystream independently of both the cipher text bitstreams and plaintext. The single-bit error in the cipher text would only produce a single bit error in the decrypted text. Please refer to *Figure 2.8* for the flowchart.

Equation 2.8: Output Feedback (OFB) Encryption Equation

$$I_1 = IV;$$

$$I_j = O_{j-1}$$

$$O_j = CIPH_K(I_j)$$

$$C_j = P_j \oplus O_j$$

$$C^*{}_n = P^*{}_n \oplus MSB_u(O_n)$$

Equation 2.9: Output Feedback (OFB) Decryption Equation

$$I_1 = IV;$$

$$I_j = O_{j-1}$$

$$O_j = CIPH_K(I_j)$$

$$P_j = C_j \oplus O_j$$

$$P^*{}_n = C^*{}_n \oplus MSB_u(O_n)$$

C = ciphertext, *P* = plaintext, *CIPH$_K$(X)* = the forward cipher function, *j* is the second data of the sequence from left to right, I$_1$ is the first data from the sequence from the Input, *X⊕Y* bitwise exclusive-OR of two strings X and Y must be of the same length, *IV* is the Initialisation vector, *I* is the input block, *MSB* is the Most Significant Bit, *O* is the output, *P** last block of the plain text, *n* is the number of data blocks or data segments in the plaintext, and *u* is the number of bits in the last plaintext or ciphertext block.[21]

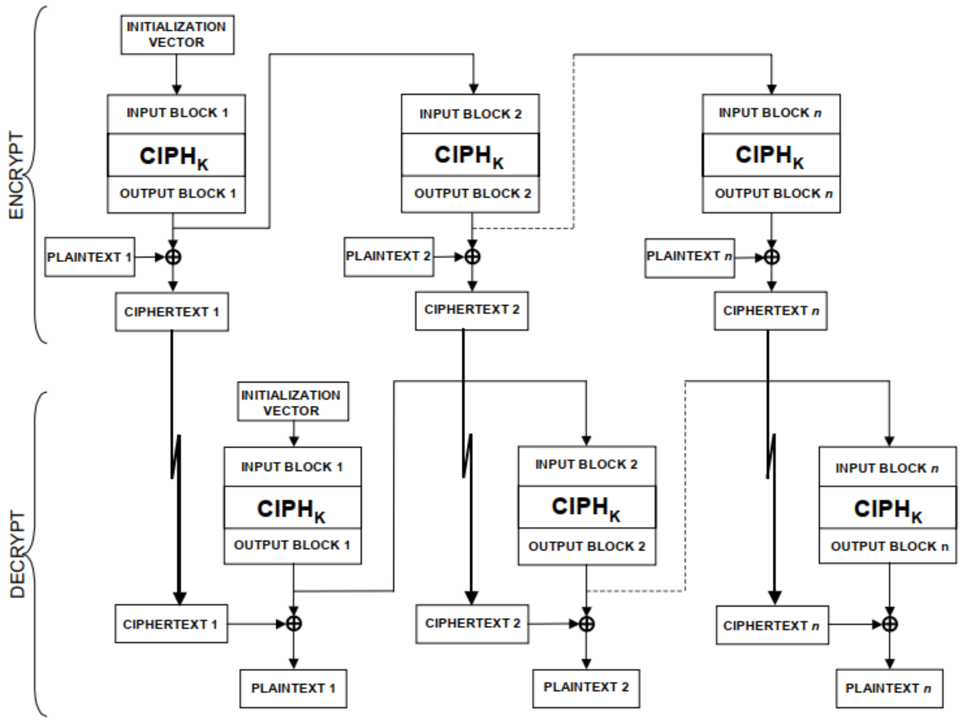


Figure 2.8: OFB encryption and decryption flow chart, taken from "Recommendations for Block Cipher Modes of Operation Methods and Techniques" [21]

Counter (CTR) mode operates on the blocks as a stream cipher and serves on the individual block like ECB. However, CTR uses various key inputs to unique blocks to prevent the same ciphertext from occurring. Additionally, each block of ciphertext has a specific location within the encrypted message. Then, CTR mode allows blocks to be processed in parallel, therefore offering performance advantages when parallel processing and multiple processors are available. It is not susceptible to ECB's brute-force, deletion, and insertion attacks. Please refer to *Figure 2.9* for the flowchart.

Equation 2.10: Counter (CTR) Mode Encryption Equation

$$O_j = CIPH_K(T_j)$$

$$C_j = P_j \oplus O_j$$

$$C^*{}_n = P^*{}_n \oplus MSB_u(O_n)$$

Equation 2.11: Counter (CTR) Mode Decryption Equation

$$O_j = CIPH_K(T_j)$$

$$P_j = C_j \oplus O_j$$

$$P^*{}_n = C^*{}_n \oplus MSB_u(O_n)$$

C = ciphertext, *P* = plaintext, $CIPH_K(X)$ = the forward cipher function, *j* is the second data of the sequence from left to right, $X \oplus Y$ bitwise exclusive-OR of two strings X and Y must be of the same length, *MSB* is the Most Significant Bit, *O* is the output, P^* last block of the plain text, *n* is the number of data blocks or data segments in the plaintext, and *T* is the counter block.[21]

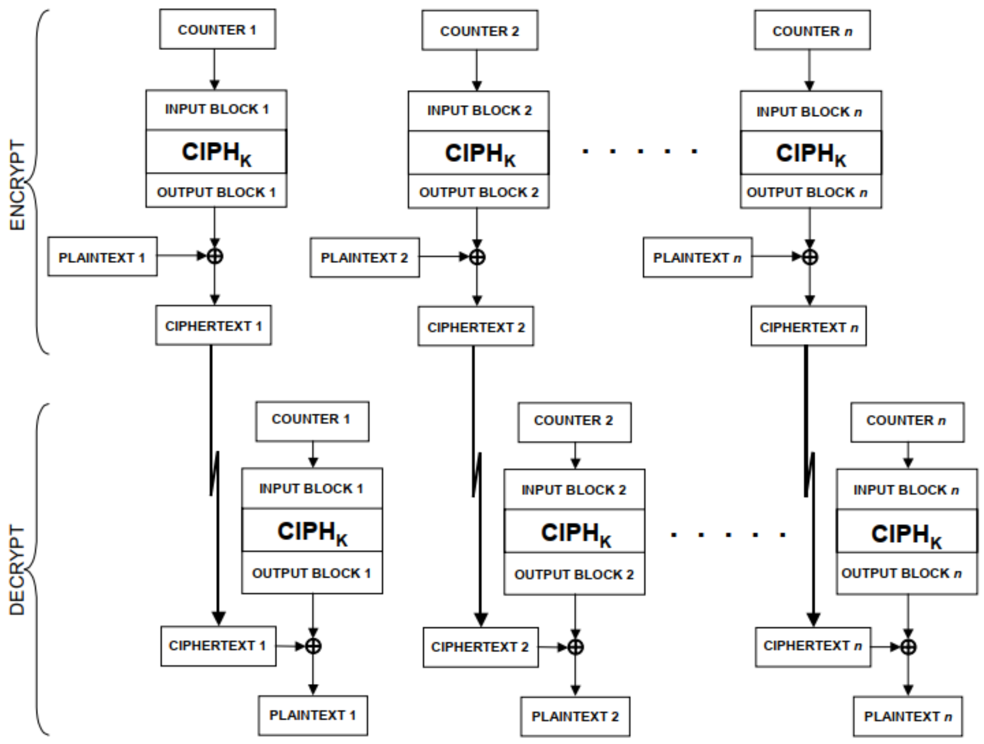


Figure 2.9: CTR encryption and decryption flow chart, taken from "Recommendations for Block Cipher Modes of Operation Methods and Techniques" [21]

2.1.2.4 Public key cryptography (PKC)

Public key cryptography (PKC) was first publicised in, 1976 at Stanford University by Professor Martin Hellman and graduate student Whitfield Diffie. PKC depends on mathematical functions that are easy to compute, but the operation's inverse would be challenging to calculate; these produce a one-way process. [22] In essence, PKC uses keys, Public keys that can be shared widely and a Private key that is only known to the owner. Therefore, using the receiver's public key, anyone can encrypt a message, but the message's decryption is achieved only using the receiver's private key.

2.1.2.5 Cryptographic Hash Function (CHF)

Cryptographic Hash Function (CHF) is a mathematical algorithm that maps data of a message's arbitrary size to a string of bits of a fixed size (this is called the hash), a one-way function. One of the vital properties of CHF is that the output of the process must look random; this is determined by a series of behaviours and conditions that the CHF must satisfy:

- First preimage resistance: Essentially, given an arbitrary hash code, it is computationally impossible to find an input that the hash function maps to that hash code (one-way function).
- Second preimage resistance: Like the first preimage resistance as given input to the hash function, it would be computationally impossible to find a second input that provides the same hash code.
- Collision resistance: When a hash function is collision-resistant, it is computationally impossible to find two inputs that give the same hash code.
- Indistinguishable from random: A hash function is indistinguishable from a random process if an attacker cannot tell the difference between the hash function and a function chosen entirely at random from all the tasks with the same input/output characteristics as the hash function. A hash function can only be indistinguishable from random if there is a key.[23]

Due to the CHF properties of looking random, the slightest change of a letter in a word would result in drastic changes in the output result.

2.1.3 Polymorphism

Polymorphism has multiple denotations according to the various industries and fields. For this research, both the information technology industry and the biological field will cross-pollinate, as human immunology translates to a novel security system for IoT devices.

In biology, polymorphism is the "*discontinuous genetic variation which results in the occurrence of several different forms or types of individual among the members of a single species*"[24]. These variations do not alter the individuals into sub-species from the original. An example would be the different blood types in humans and the smooth graduation of heights among the human population. Please note that if there are many polymorphic variations within a species, it can persist over many generations, especially if the variation is advantageous in natural selection. Therefore, some polymorphic variation could be advantageous to a species' survival.

Humans have an innate and adaptive immunity in terms of immunology. Innate immunity responds and recognises generic targets on foreign agents (pathogens), whereas adaptive immunity recognises specific targets using "*randomly generated receptors that have a virtually unlimited recognition repertoire*"[25]. There are two different types of adaptive immunity called humoral immunity and cell-mediated immunity, and please see *Figure 2.10* for a visual representation. The properties of adaptive immune responses are specificity, diversity, memory, clonal expansion, specialisation, contraction and homeostasis, and nonreactivity of self[2].

Table 2: Properties of adaptive immune responses according to Functions and Disorders of the immune system[2]

Feature	Function
Specificity	To ensure the targeted foreign agents (antigens[1]) get the appropriate responses
Diversity	To have a large variety of responses for a large variety of antigens
Memory	To know which response was the most effective in protecting against future exposure to the same foreign agent
Clonal expansion	Increases the number of antigen-specific lymphocytes[2] from a small number of naive lymphocytes, allowing for a higher concentration-response as more cells are aware of the antigen
Specialisation	Specialisation gives the ability to generate a more specific defence against different microbes and antigens
Contraction and homeostasis	The ability to respond to an unknown or newly encountered antigen
Nonreactivity of self	Not to harm the host during the response to an antigen.

[1] Any type of toxin or foregin substance that induces a immune response in the body.
[2] B Lymphocytes produce proteins called antibodies which mediates an humoral immunity response.

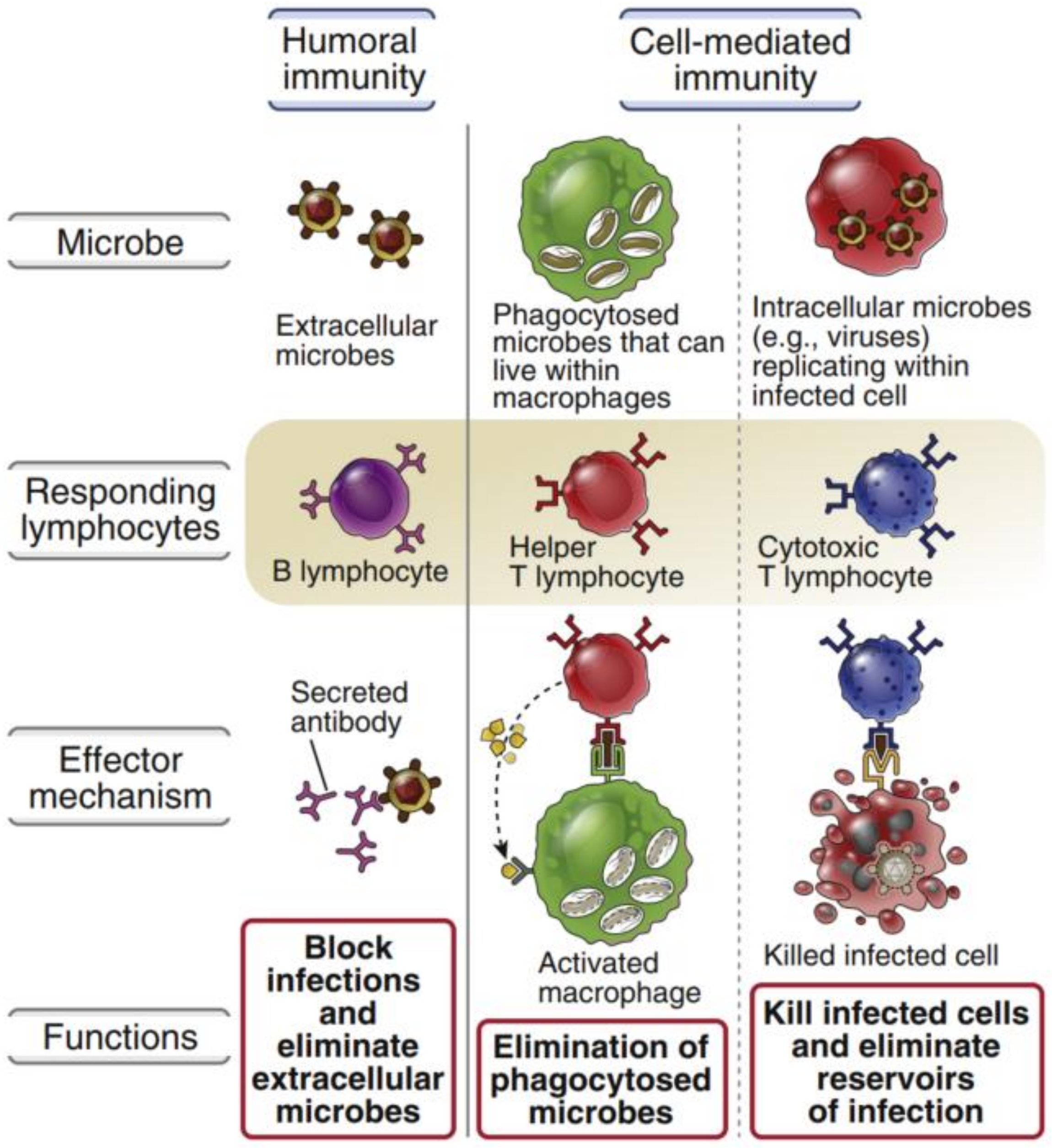

Humoral immunity
Cell-mediated immunity
Microbe
Extracellular microbes
Phagocytosed microbes that can live within macrophages
Intracellular microbes (e.g., viruses) replicating within infected cell
Responding lymphocytes
B lymphocyte
Helper T lymphocyte
Cytotoxic T lymphocyte
Effector mechanism
Secreted antibody
Activated macrophage
Killed infected cell
Functions
Block infections and eliminate extracellular microbes
Elimination of phagocytosed microbes
Kill infected cells and eliminate reservoirs of infection

Figure 2.10: Types of adaptive immunity - image taken from[2]

To translate and reverse engineer the biology of adaptive immunity into the focus of cybersecurity, there needs to be a breakdown comparison of each of these features to the next best thing in the cybersecurity industry. Specificity ensures the targeted antigens get the correct response from antivirus programs, and they scan files and code to see if it has a similar signature to known malware and viruses. If a known virus is detected, then the directed response will quarantine the file and then delete the program[26].

The function of diversity, memory and specialisation is the adaptive immune system's ability to respond to different antigens and remember which responses are most responsive and efficient. In a nutshell, this is also known as a threat database in the cybersecurity field. The threat database aims to log current virus events with their signatures and repel those viruses from computer systems.

Clonal expansion is the adaptive immune system's ability to alert nearby cells to an active virus. For an electronic system to imitate clonal expansion, each device needs to be within an interconnected network like a mesh network, server and client or other connected topologies to allow the data of threats to be shared with all connected devices. The advantage of having all connected devices aware of a virus in the system is to reduce its spread and take appropriate measures according to pre-written protocols.

Usually, an infected machine is isolated and quarantined from the rest of the system to avoid malware from reproducing and spreading. Stopping this spread is the main feature of the contraction and homeostasis function, which aims to respond to unknown or newly encountered antigens. The difficulty with unknown viruses with minimal data would be the severity of the issue they pose to any given system. Therefore, the best approach is to contain the virus and then study it as a signature to see if any other systems have the same virus.

Lastly, the ability to be nonreactive to oneself during the response to an antigen is a crucial application in any security system. Suppose an interconnected system, such as a server with many clients, is infected. In that case, the best approach, depending on the severity of the issue, might limit communication streams to non-infected clients to avoid further spread and then to remotely factory reset the infected clients to a previous image of the system that does not have a virus. Additionally, suppose a scheduled backup is a commonality within a system. In that case, the recovery of the most recent backup that does not have the virus is the best option to restore the client's functionality and eject the system's virus.

For programmers and engineers alike, adaptive immunity seems like an intrusion detection system with a database for known security risks. The detection of a foreign agent that correlates to a known security risk can then be subject to a specifically targeted response. It is also relatively common to block known incoming threats by either allowing specific sites, applications or communication streams or blocklisting threats known to the host/organisation.

Polymorphism within computer science and programming comes from the notion of type theory, which is a system that every term has a type that defines the operations, meanings and how it performs[27]. An example of type theory in programming is when a name (variable) might denote different instances of many different classes as long as they have a common superclass. Therefore, any object denoted by the same variable can respond to a standard set of operations differently. The expected output through various operations; this type of polymorphism is called inclusion polymorphism[28].

Another primary class of polymorphism is ad hoc polymorphism, described by Strachey[29], who stated that symbols used in programming, such as plus (+), could also be defined to mean different things. This concept is also known as overloading. Object-oriented programming languages like C++ could declare functions having the same names, as long as they have different invocations. For example, there could exist two functions (also known as methods or subroutines) called “Addition”, whereby one of these functions could return an integer by taking in two integer variables as parameters and adding the integer value (2+2 will return 4). However, the other function called with the same name can return a string by taking in two string variables from the parameter and concatenating the values as a return statement (“hello,”, “there” will return “hello, there”).

Lastly, parametric polymorphism allows a data type or a function to be written generically; therefore, it can handle values uniformly without depending on their data type[30]. Handling values uniformly allows the language to be more expressive while maintaining the complete static type-safety to prevent type errors.

Nonetheless, polymorphism within the cybersecurity field is an amalgamation of both how biological viruses mutate and the different classes of polymorphism. In cybersecurity, polymorphic viruses are a self-replicating piece of code that uses a polymorphic engine (also called a mutation engine) to mutate while keeping the algorithmic output result intact. There are two ways a polymorphic engine works: either encrypting the code by using a cypher to make it harder for antivirus systems to detect the signature of the virus through the code, or by using an obfuscation technique to prevent tampering, reverse engineering, and detection of its true purpose.

This research aims to produce a security system with human immunology’s logic operations while deterring and deceiving invasive unauthorised clients using polymorphic techniques such as inclusion, ad hoc, and parametric polymorphism to protect approved clients within the network system. The next chapter will propose a framework for how a polymorphic security system is implemented on commercial IoT devices, therefore tackling intrusive attacks and working within microcontrollers’ limited processing power.

Furthermore, the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS) draws conceptual inspiration not only from immunology but also from neurophysiology. In biological organisms, particularly in the human brain, different states of activity are characterised by varying frequencies of brain waves: Beta waves dominate during active thinking and stress, while Delta waves are predominant during deep sleep. Similarly, the AABCMS dynamically transitions the IoT device between high-processing (active) and low-power (sleep) states based on real-time battery health predictions. This parallels how biological systems conserve energy during periods of low activity and expend more resources during critical functional demands. Such bio-inspired adaptation enables IoT devices to maximise operational longevity while ensuring readiness to respond when necessary, thereby offering a novel paradigm for battery-aware security systems.

2.1.4 Battery Management in IoT Security Systems

Despite extensive research into IoT security protocols, comparatively limited attention has been given to the intersection of battery management and secure communication frameworks in IoT environments. Existing lightweight cryptographic approaches, while optimised for computational efficiency [31]. Often fail to account for the dynamic energy profiles of devices operating under variable loads and environmental conditions. Studies such as [32] highlight that energy-aware communication protocols have been proposed independently of security considerations, yet comprehensive frameworks that synchronise cryptographic adaptability with battery curve monitoring remain scarce. This gap motivates the development of the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS), which uniquely combines real-time battery health prediction, adaptive encryption strength selection, and sleep cycle optimisation to maintain the operational longevity and security resilience of IoT devices. By integrating power management directly into the core security model, this system extends beyond conventional designs that treat energy management and cybersecurity as isolated concerns.

While traditional IoT energy management strategies often focus solely on static sleep intervals or power gating techniques, the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS) advances the paradigm by drawing conceptual inspiration from biological neural systems. In particular, human brainwave activity provides a relevant analogue: different cognitive and physiological states are associated with distinct electrical patterns. Beta waves (13–30 Hz) dominate during active thought, stress, and problem-solving, associated with heightened glucose consumption and energy demand [33], whereas Delta waves (0.5–4 Hz) predominate during deep sleep phases, associated with significant energy conservation and restoration [34].

The AABCMS dynamically transitions IoT devices between high-processing active modes and low-power sleep states according to real-time battery health predictions. This mirrors how biological systems strategically allocate energy, upregulating activity during high cognitive load and downregulating it during rest. In the context of IoT systems, such dynamic adaptation allows devices not merely to conserve battery life but to operate with biological-like resilience, maintaining operational readiness during critical demands while minimising unnecessary energy expenditure during idle periods. By integrating bio-inspired adaptive behaviour, the AABCMS ensures optimised longevity without compromising functional responsiveness, positioning it as a novel contribution to energy-aware security system design.

2.2 Focus on Research and Evaluating Studies

This sub-chapter aims to go over a few related studies to discuss their findings, appreciate their contributions, and see how they can benefit this thesis.

“Tree-Chain: A fast lightweight consensus algorithm for IoT application”[35] proposes a new way to implement blockchain on low processing power devices such as IoT devices. The main reason blockchain is challenging to implement on IoT platforms is the expensive computational power required for the validation and consensus algorithms which would cause significant transaction delays between clients. This paper, written by Ali Dorri and Raja Jurdak, proposes a scalable tree-chain for fast blockchain installation that introduces two randomisation levels in the validator. One of the randomisation levels are in the transaction level, where the validator of each transaction is selected randomly based on the most significant characters of the hash function output (known as consensus code), and the other in the blockchain level where the validator is randomly allocated to a particular consensus code based on the hash of their public key. The authors also implemented the tree-chain to work with parallel chain branches; therefore, each validator’s corresponding transactions are in an individual ledger. The paper gives a novel solution for blockchain within IoT platforms that will allow low-power processors to perform blockchain interactions.

Additionally, due to the blockchain implementation, there were a few security benefits against attacks, such as a denial of service attack. The denial of service attack (DOS) as the validators in the network monitors the cumulative number of transactions generated within a particular consensus code range and the number of such transactions. If this threshold is surpassed, the validators will choose a new validator for the corresponding consensus code. Hence, the DOS was ineffective for this tree-chain approach. A tree-chain approach would prove fruitful for the lightweight polymorphic security system as it integrates a shared ledger between the server and client to share predetermined encryption keys and passwords for approved clients while being effective against common attacks like DOS.

Effy Raja Naru et al. wrote a paper to review recent lightweight cryptography in IoT[36]. Though a short paper, the table of comparison between lightweight cryptography related works in IoT is beneficial as it gives pros and cons for each technique used. However, the paper does argue a challenge that all of these lightweight cryptography techniques have in common, which is the physical security for IoT. The notion of physical security and hardware trojans is a common threat against IoT devices; therefore, some protection against these types of attacks is considered in this thesis, especially in the testing phase.

Directly related to the previous paper, “Hardware-assisted Cybersecurity for IoT devices”[37], it puts forth a solution to IoT’s hardware-related cybersecurity issues on top of the software only solutions to layer the security for IoT devices. This study’s hardware-based methods give leverage to the hardware modules by collecting micro-architecture information to analyse prevailing software level threats and vulnerabilities. Some of the hardware-based methods include a runtime micro-architectural event monitor, side-channel information, trusted platform modules, and a security-aware design. This study’s most exciting integration is the discreet trusted platform module chip, which allows cryptographic keys tied to specific platform measurements and are protected from disclosure to any untrusted hardware components, processes, or software. Such hardware implementation is manufactured by companies like Intel’s Software Guard Extention[38] and ARM with their TrustZone chip[39] that can be incorporated into existing modules during the development phase.

Further exploring hardware related subjects, a review article by Mingfu Xue et al. called “Ten years of hardware Trojans: a survey from the attacker’s perspective”[40] gives invaluable insight into how hardware trojans were executed in various stages of the product lifecycle. There are five stages within the supply chain: design, synthesis and verification, fabrication, testing, and distribution, where the attackers have opportunities to attack. In the design phase, attackers (in-house design team attackers) can use flexible methods to implement any malicious function and create a side-challenge/covert communication channel leakage. In the synthesis and verification stage, the attackers will be 3PIP vendor attackers who implement malicious functions with flexibility by modifying the IP design at the RTL netlist or other specification levels. Fabrication stage attackers try to change the netlist in the layout or modify the manufacturing process. During the testing phase, the attackers directly modify the hardware trojan’s detection results or modify the test data to mislead the detection results. Lastly, in the distribution phase, the reverse- engineering a chip to pirate the design or directly replace it with a trojan inserted version during the transportation.

Designing with a vision for the future helps devices be in service for longer, as the device can be upgraded through software and modified by adding additional hardware. The paper “lattice-based cryptography for IoT in a quantum world: are we ready?”[41] shows how current FPGA’s and other IoT platforms could implement lattice-based cryptography. In order to implement lattice-based cryptography in an IoT platform, there are a few challenges. One of the challenges is communication bandwidth, as most embedded processors are memory-constrained, therefore only suited for minor security parameters, such as IoT applications with limited transmission bandwidth (through Wi-Fi). Security strength often balances performance and security, as any brute-force cryptanalytic efforts require more computational resources, increasingly for the required search on a block cipher such as AES-128 and other similar ciphers. However, the trade-off between performance and the required security is generally less desirable due to the associated overhead. This paper gives an excellent overview of what IoT devices are lacking. However, it does open a Pandora's box to a more significant impact on quantum computing and its ability to crack modern security systems with ease.

One of the challenging issues in IoT devices is the tampering of firmware, as it is challenging to detect and recover from the tampered firmware. The paper “ChainVeri: blockchain-based firmware verification system for IoT environment”[42] proposes a new blockchain-based firmware verification system that used a shared palette (ledger) to check the devices to see any firmware tampering have been made. The palette comprises the block header, which encompasses the block hash, block size, block version, previous block hash, time, difficulty, and nonce. The palette also has another module called the verification information, verifying the device model, firmware version, verifier, and identification. Having these functions in the palette allows the blockchain to know the block’s version when the structure of the palette changes. It can also verify the device through a universally unique identifier and check and verify its firmware.

Due to IoT devices and storage’s limited computational power, traditional off-the-shelf solutions are resource contained on these devices and therefore not recommended for security implementation. The paper “Pseudo-Random Number Generator and Hash Function for Embedded Microprocessors”[43] puts forth a solution to overcome this problem by implementing lightweight techniques for efficient Pseudo-Random Number Generator and Hash function to reduce memory consumption and accelerate performance. The reason behind using a pseudo-random number generator is to build and generate harder to break keys and other secret parameters on embedded processors. This paper does propose a relatively efficient way to harden existing security methods such as AES by implementing a pseudo-random number generator; therefore, not too much change needs to occur on the broader system for integration.

“Lightweight Cryptography Algorithms for resource-constrained IoT devices: A Review, Comparison and Research Opportunities”[44] is an excellent paper that provides a holistic view and compares various lightweight cryptography algorithms that are available in the market. The critical common challenges with conventional IoT based cryptography are limited memory (RAM, ROM, and registers), limited computational power, the small surface area of the device, lower battery power (no battery power with RFID tags), and real-time operations. The way this paper compares the different cryptography algorithms is by three characteristics: physical, which is the physical area, memory and battery power. Performance, which determines the computing power (latency and throughput); lastly, the security characteristic measures the minimum security strength in bits, attack models, and side-channel and fault-injection attacks. With these various parameters, the thesis can benefit by paying closer attention to what is valued in this paper's comparisons of different lightweight cryptography algorithms.

There is a comprehensive and detailed thought for what type of IoT devices should be implemented in an IoT platform in any IoT system development. Therefore, a study compares and “Reviews Low-End, Middle-End, High-End IoT devices”[45]. The way the IoT devices are classified are as follows: Low-end devices (such as at ATTINY85) are classified as having less than 50kB of RAM, less than 250kB of Flash, devices that do not support an RTOS to devices with RTOS, communication protocols range from gateway communication, lightweight protocols such as Constrained Application Protocol (CoAP), and communication protocols such as HTTP. Lastly, security vulnerabilities whereby data is compromised, causing a medium to a high threat. Middle-End devices (like the ESP8266 and ESP32) provide more outstanding features and processing capabilities such as more RAM, Flash, higher clock speeds, and various communication protocols like Wi-Fi, Bluetooth and Bluetooth Low Energy (BLE). High-end devices (Raspberry Pi, PandaBoard, HummingBoard, and more) typically are single-board computers with powerful processing units and plenty of RAM to provide a graphical user interface or even the ability to run custom operating systems like Windows 10 IoT, Ubuntu, Linux, and Raspberry OS. The main reason why this study is essential for this IoT development is that it helps developers to decide which board to use for a particular functionality. Such as if there is a requirement to gather air pollution data in a given environment, there isn’t a need to use a high-end device as they are more expensive for scaling up, instead of using a middle-end device would prove fruitful as they have enough processing power and a communication protocol to gather the data and send it to a server.

3.0 Chapter Three: Background

Chapter three covers the history and theory of operations on the various topics prevalent in this research, such as IoT boards, Cloud computing, Cryptography, immunology and blockchain.

3.1 Current IoT boards

There are thousands of IoT boards in the market that use various microprocessors and expansion modules to enable connectivity. However, when looking for the correct IoT board for a project, the consumer must first understand the project requirements and then find an appropriate board to continue development for either personal use or a client. Typically when researching the appropriate board for an IoT based project, there are a couple of things to keep in mind. The main hardware selection criteria are summarised in Table 3 for different features and why they are essential when designing an IoT product.

Table 3: Key features of IoT devices that contribute to the design of an IoT device

Feature	Reasoning
Clock Speed	The clock speed is how fast the data is transferred between the microprocessor and between the microprocessor and memory, which must be synchronised.[46] The importance is that the faster the clock speed, the higher the power consumption; therefore, for mobile IoT devices, it is critical to lower and slow down the clock speed to improve the battery lifetime. A similar approach is also achieved by using deep sleep/sleep modes.
Power Consumption	Most IoT devices are connected directly to a constant power source. However, it is increasingly common to find more IoT devices that are mobile and battery-powered. There are many aspects that contribute to power consumption, from clock speed, the microprocessor, type of connectivity, sleep modes, effective programming (e.g. how many cycles it takes to do specific functions), and the number of connected sensors.
Connectivity	How does the IoT board transfer data between all points of the ecosystem? Does it use short-range communication such as Bluetooth Classic, Bluetooth Low Energy (BLE), or Radio Frequency Identification (RFID)? Mid-range communication traditionally falls under Wi-Fi, but Zigbee, MQTT and other mesh protocols for home automation are also common. For long-range communication, it is typical to use Low Power Wide Area Networks (LPWANs) like NB-IoT, LTE-M, LoRa, Sigfox, and cellular connections.
Communication	What communication protocols (I/O interfacing) does the microprocessor use? UART, USART, USB, I2C, SPI, CAN, and so on are crucial as some sensors can only transfer data using specific protocols.

The common IoT manufacturers are usually well-established manufacturers who typically sell the same microcontrollers to different board designers who make specific selling points. The common manufacturers for IoT microcontrollers are ARM with their Cortex series[47], [48], [49], Atmel with their AVR processors[50], Espressif is leading the charge for the all in one IoT chip[51], Nordic Semiconductors [52], and Broadcom[53].

3.2 History of Cloud Technologies

In a pithy manner, cloud computing can be described as “a style of computing in which scalable and elastic IT-enabled capabilities are delivered as a service using Internet technologies.”[54]. However, a more concise definition of cloud computing from Thomas Erl is “Cloud computing is a specialised form of distributed computing that introduces utilisation models for remotely provisioning scalable and measured resources.”[55]. Cloud computing is not a new technology, as the idea was primarily developed to be a military mainframe in the 1950s to connect computer terminals across an internal matrix and have a decentralised storage technology. The term cloud computing was coined by the Compaq company in an internal document in 1996, but was popularised by Amazon.com after it released its Elastic Compute Cloud in 2006. The evolution of cloud computing is depicted in *Figures 3.1* and *3.2*.

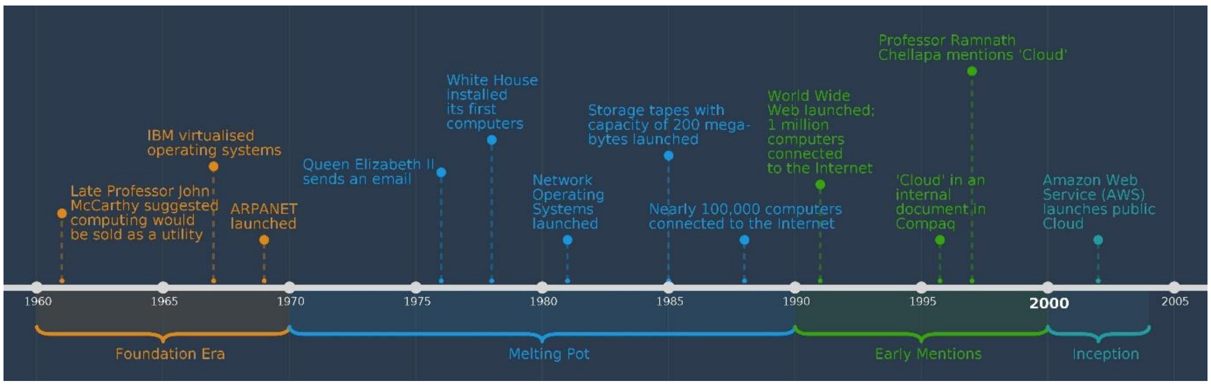

Late Professor John McCarthy suggested computing would be sold as a utility
IBM virtualised operating systems
ARPANET launched
Queen Elizabeth II sends an email
White House installed its first computers
Network Operating Systems launched
Storage tapes with capacity of 200 mega-bytes launched
Nearly 100,000 computers connected to the Internet
World Wide Web launched; 1 million computers connected to the Internet
Professor Ramnath Chellapa mentions 'Cloud'
'Cloud' in an internal document in Compaq
Amazon Web Service (AWS) launches public Cloud
1960
1965
1970
1975
1980
1985
1990
1995
2000
2005
Foundation Era
Melting Pot
Early Mentions
Inception

Figure 3.1: Early Timeline of Cloud Computing - Taken from BCS (https://www.bcs.org/content-hub/history-of-the-cloud/)[56]

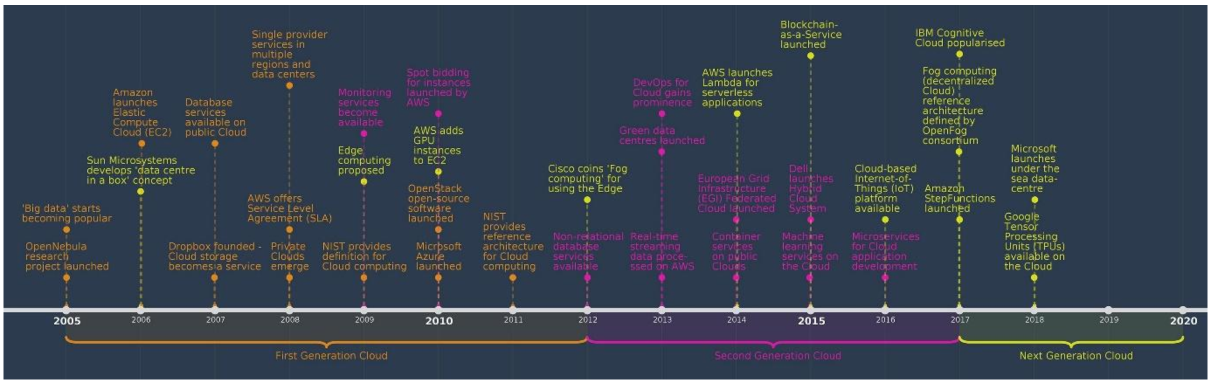

'Big data' starts becoming popular
OpenNebula research project launched
Sun Microsystems develops 'data centre in a box' concept
Amazon launches Elastic Compute Cloud (EC2)
Database services available on public Cloud
Dropbox founded - Cloud storage becomes a service
Single provider services in multiple regions and data centers
AWS offers Service Level Agreement (SLA)
Private Clouds emerge
Monitoring services become available
Edge computing proposed
NIST provides definition for Cloud computing
Spot bidding for instances launched by AWS
AWS adds GPU instances to EC2
OpenStack open-source software launched
Microsoft Azure launched
NIST provides reference architecture for Cloud computing
Cisco coins 'Fog computing' for using the Edge
Non-relational database services available
DevOps for Cloud gains prominence
Green data centres launched
Real-time streaming data processed on AWS
AWS launches Lambda for serverless applications
European Grid Infrastructure (EGI) Federated Cloud launched
Container services on public Clouds
Blockchain-as-a-Service launched
Dell launches Hybrid Cloud System
Machine learning services on the Cloud
Cloud-based Internet-of-Things (IoT) platform available
Microservices for Cloud application development
IBM Cognitive Cloud popularised
Fog computing (decentralized Cloud) reference architecture defined by OpenFog consortium
Amazon StepFunctions launched
Microsoft launches under the sea data-centre
Google Tensor Processing Units (TPUs) available on the Cloud
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
First Generation Cloud
Second Generation Cloud
Next Generation Cloud

Figure 3.2: Current Timeline of Cloud Computing - Taken from BCS (https://www.bcs.org/content-hub/history-of-the-cloud/)[56]

The National Institute of Standards and Technology (NIST) made three models for cloud-computing providers: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).[57]

IaaS, or Infrastructure as a Service, is the lowest level in the network infrastructure, encompassing physical computing resources, scaling, backups, security, memory, partitioning, and more. At the IaaS level, a hypervisor will run all the virtual machines as guests within a cloud operating system. Therefore, it can support large numbers of virtual machines and allows the users to scale up and down according to their usage and demand. The NIST definition for IaaS is for the "the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer [can] deploy and run arbitrary software, which can include operating systems and applications."[57] However, the users are not in control of the underlying cloud infrastructure, and they only control the operating system, storage, deployment applications, and have limited control over networking components.

PaaS offer the consumers a development environment to the application developers to deploy unto the cloud infrastructure, either consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the cloud provider.

Cloud providers manage the infrastructure and platform that runs the application/ software for SaaS, allowing the consumers to gain access to the application software and databases. The NIST definition is "The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure"[57] that can be accessed through various client devices, either through a web browser, programming interface, or a thin client interface.

Cloud infrastructure is the collection of hardware and software which enables the five essential characteristics of cloud computing, shown in Table 4. The cloud infrastructure contains both the physical and abstract layers (in that order, as the abstraction layer sits on top of the physical layer). The physical layer consists of the hardware resources essential for the cloud service, such as the server, network components and storage. The abstraction layer consists of the software deployed across the physical layer, which gives the cloud its characteristics.

Table 4: Essential Characteristics of Cloud Computing[57]

Essential Characteristics of Cloud Computing	
Characteristics	**Definition**
On-demand self-service	A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.
Broad network access	Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).
Resource pooling	The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources, but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.
Rapid elasticity	Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear unlimited and can be appropriated in any quantity at any time.
Measured service	Cloud systems automatically control and optimise resource use by leveraging a metering capability (usually pay-per-use) at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilised service

3.3 Cryptography

Cryptography, fundamentally, is the science of encoding and decoding information to maintain confidentiality, integrity, and authenticity in communication systems [58]. Historically, cryptography dates back to ancient times, with primitive encryption techniques, such as the Caesar cipher, used predominantly in secure military communications [59]. Modern cryptographic methods are broadly categorised as symmetric-key cryptography, asymmetric-key cryptography, and cryptographic hash functions [60]. In symmetric cryptography, a single private key is shared between the sender and receiver to encrypt and decrypt data, offering computational simplicity and speed. However, the challenge with symmetric encryption lies in secure key distribution and management [61]. Asymmetric cryptography, conversely, employs a pair of keys (public and private), eliminating the key distribution dilemma by openly distributing the public key, yet requiring significantly greater computational resources due to its algorithmic complexity [60].

In the context of the Internet of Things (IoT), conventional cryptographic algorithms, designed primarily for traditional computing environments, often become computationally prohibitive due to hardware limitations and stringent power constraints [62]. Consequently, lightweight cryptography has emerged as a pivotal domain specifically tailored for resource-constrained devices prevalent within IoT ecosystems. Lightweight cryptographic algorithms are optimised to operate efficiently on limited hardware, low memory footprints, and minimal processing power while providing a satisfactory security threshold suitable for most IoT applications [63]. Notable lightweight encryption algorithms include SPECK and SIMON, both block cipher algorithms developed by the United States National Security Agency (NSA) to ensure robust security while reducing computational overhead on constrained devices [64]. Such lightweight algorithms effectively balance between cryptographic strength and resource efficiency, enabling integration into microcontroller-based IoT devices such as ESP32 and ATMEGA328P, as demonstrated in this thesis.

Adaptive cryptography further extends the concept of lightweight cryptography by dynamically adjusting cryptographic strategies in response to evolving environmental conditions, threat landscapes, and resource availability [65]. This dynamic adaptability is critical for IoT devices, given their exposure to varying operational contexts and constrained battery resources. Adaptive cryptographic methods assess device-specific parameters such as battery voltage, processor cycles, and available memory to select the optimal encryption method that ensures security without exceeding energy budgets [66]. This methodology parallels biological systems, notably the adaptive immune system, which dynamically adjusts defensive strategies in response to varying pathogen threats [67].

In the presented Bio-Inspired Lightweight Polymorphic Security System for IoT devices, adaptive cryptography is embedded within the Adaptive Amoeba Complexity subsystem, where encryption strength dynamically varies depending upon real-time device battery levels and processor constraints. When the battery level is high, computationally robust block cipher algorithms, such as SPECK and SIMON, can be implemented to maximise security. Conversely, as battery power decreases, computationally simpler encryption methods, such as XOR, Caesar cipher, or ROT13, are selected to conserve energy resources while still maintaining baseline encryption standards. Thus, the system ensures an optimal trade-off between security strength and power consumption, underpinning both operational efficiency and security resilience within resource-limited IoT environments.

3.4 Immunology

Immunology is the branch of biomedical science concerned with the study of the immune system, its physiological functioning in health, and its malfunctions in disease states. At its core, the immune system is a highly dynamic, adaptive, and decentralised network that protects the organism against pathogenic threats by distinguishing between self and non-self entities [67]. This selective and rapid-response capability forms the fundamental biological inspiration for the design of the Bio-Inspired Lightweight Polymorphic Security System for IoT devices presented in this thesis.

The immune system is traditionally divided into two major branches: the innate immune system and the adaptive immune system [68]. The innate immune system provides the first line of defence, characterised by non-specific, immediate responses against invading organisms. Mechanisms such as macrophage phagocytosis, neutrophil activation, and natural killer (NK) cell-mediated cytotoxicity represent innate immunity’s swift but generalised approach [69]. By contrast, the adaptive immune system demonstrates specificity and memory, capable of tailoring immune responses to particular antigens via the generation of antigen-specific B and T lymphocytes [70]. A hallmark of adaptive immunity is immunological memory, whereby previous encounters with pathogens are "remembered," allowing for faster and stronger responses upon subsequent exposures.

Drawing inspiration from these biological principles, the proposed polymorphic IoT security framework emulates key immune system functionalities. Auto-detection of clients mirrors innate immune surveillance, wherein every incoming client connection is rapidly assessed against a predefined list of approved clients (analogous to the immune system's recognition of self-antigens). Clients that fail initial authentication undergo a secondary scrutiny process analogous to antigen presentation and adaptive immune activation. If identified as foreign or untrustworthy, the system initiates an auto-ejection response akin to the cytotoxic response mounted against pathogens.

Furthermore, the shared ledger mechanism within the security system is comparable to immunological memory, maintaining a dynamic record of approved credentials to enable swift reacquisition of trusted clients and facilitate prompt defence against desynchronised or rogue agents.

At a molecular level, biological immune systems employ receptor diversity and mutational adaptability to counter evolving pathogenic threats [71]. Similarly, the polymorphic security system dynamically alters encryption methods, session identifiers, and access credentials upon detecting anomalous behaviour. This process ensures that even if an attacker compromises one instance of the security environment, future iterations become increasingly difficult to predict or penetrate, reflecting the adaptive immune system's principle of clonal expansion and somatic hypermutation [72].

In mathematical terms, the diversity generated by the immune system can be conceptualised by calculating the possible combinations of antigen receptor genes through combinatorial joining and junctional diversity, estimated to be in the order of 10121012 (10^{12}) unique receptors [73]. Analogously, in the proposed system, the dynamic rekeying process ensures a large cryptographic search space, increasing exponentially with each adaptive response cycle and significantly enhancing system resilience. The polymorphic transformation of encryption methodologies can be expressed mathematically as:

Equation 3.1: General Formula for Polymorphic Transformation of Encryption Methodologies

$$S_{new} = f(S_{old}, E_k)$$

Where S_{new} represents the updated session environment, S_{old} is the existing session structure, and E_k is a dynamically selected encryption transformation function from a set of polymorphic candidates.

The adaptive mechanisms of the immune system serve as an elegant and efficient model for cybersecurity in highly dynamic and resource-constrained environments such as the IoT. By applying immunological principles of detection, ejection, and memory to the engineering of IoT security, this research bridges interdisciplinary concepts and presents a novel framework for achieving autonomous, energy-efficient, and resilient security architectures.

3.5 Blockchain

Blockchain technology is a decentralised and distributed ledger system designed to record transactions securely, transparently, and immutably across a network of participants [74]. Initially conceptualised as the foundational architecture behind Bitcoin by Nakamoto [75], Blockchain has since evolved into a powerful general-purpose framework applicable to a wide range of domains, including finance, healthcare, supply chain management, and, increasingly, cybersecurity for Internet of Things (IoT) environments.

At its core, blockchain operates by aggregating transactions into blocks, which are then cryptographically linked to form a continuous chain. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data. The chaining mechanism ensures the integrity and immutability of the historical record: altering any single block retroactively would require the alteration of all subsequent blocks, an endeavour computationally infeasible for large networks [76].

The consensus protocols employed within blockchain systems, such as Proof of Work (PoW) or Proof of Stake (PoS), serve as trust mechanisms among otherwise untrusted entities [77]. These consensus algorithms guarantee that all participants reach agreement on the state of the ledger without the need for a centralised authority, thus enabling secure decentralisation. Mathematically, blockchain's resilience can be represented through Byzantine Fault Tolerance (BFT), where a system of 3000 nodes remains reliable if fewer than 1000 (up to one-third) of the nodes are compromised [78].

In the context of this thesis, the principles of blockchain have inspired the design of the Shared Ledger Process (SLP) within the Bio-Inspired Lightweight Polymorphic Security System for IoT devices. The ledger maintains synchronised, tamper-evident records of authorised clients, their credentials, encryption methods, and session histories. Much like blockchain nodes maintain a copy of the distributed ledger to ensure consensus and detect anomalies, each client device synchronises its local ledger upon connection or after a detected security event, ensuring consistency and integrity across the network.

The dynamic update and propagation of security credentials through the shared ledger can be formally modelled as:

Equation 3.2: General Formula for Dynamic Update and Propagation of Security Credentials through Shared Ledgers

$$L_{n+1} = H(L_n \parallel C_n)$$

where L_{n+1} represents the new ledger state, L_n is the previous ledger entry, C_n is the new client credential data, and H denotes a cryptographic hash function ensuring data integrity. This approach ensures that any tampering or desynchronisation becomes immediately apparent, akin to the cryptographic immutability principle in blockchain systems.

Moreover, just as blockchain systems possess the ability to fork and re-synchronise in the event of network partitions or attacks, the proposed framework includes a desynchronisation detection and recovery mechanism via the Adaptive Honeypot Response. In cases where legitimate clients lose synchronisation (e.g., through power failure or malicious interference), the system transitions into a controlled honeypot environment, allowing only verified UUIDs to retrieve updated ledger data while isolating potential threats.

The lightweight nature of the proposed shared ledger, tailored for resource-constrained IoT devices, represents a crucial departure from traditional blockchains, which typically involve high computational and storage overhead. By optimising the ledger entries to essential security information and using efficient hash-based authentication rather than full consensus protocols, the system balances security, performance, and energy efficiency.

The conceptual parallelism between blockchain's decentralised integrity model and the security framework presented herein thus provides a robust, scalable, and resilient foundation for autonomous, self-healing IoT security systems in heterogeneous environments.

3.6 Adaptive Energy Management: Inspiration from Neural Dynamics

As IoT devices become increasingly ubiquitous in critical applications, from healthcare monitoring to industrial control systems, the demand for optimised energy management alongside robust security has intensified. Traditional battery management systems focus predominantly on voltage monitoring and simple low-power state transitions [79]. However, these approaches often lack integration with the security frameworks of devices, leading to inefficiencies where cryptographic operations drain battery reserves unpredictably [80].

The Adaptive Amoeba Battery Curve Mapping Management System (AABCMS) developed in this research addresses this limitation by fusing battery health prediction, dynamic energy mapping, and adaptive sleep modes with real-time security communication protocols. Through techniques such as quadratic curve fitting to battery voltage profiles and dynamic adjustment of encryption complexity based on available energy reserves, AABCMS offers a biologically inspired, fully integrated model. This ensures not only the prolongation of device lifespan but also sustained operational integrity under stringent energy constraints, a necessity for the next generation of resilient IoT deployments.

The energy management techniques employed by the AABCMS mirror biological neural systems, where operational modes adaptively shift between high-energy consumption (e.g., Beta wave activity during cognitive stress) and low-energy conservation (e.g., Delta wave activity during deep sleep), aligning technological adaptation with proven biological efficiency models [81].

The development of the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS) is informed not only by immunological models but also by the adaptive energy dynamics observed in biological neural systems. During periods of intense cognitive activity, humans exhibit Beta wave patterns, indicative of increased energy utilisation, while during states of rest or sleep, Delta wave patterns emerge, associated with reduced energy consumption[33], [34].

In parallel, the AABCMS enables IoT devices to dynamically adapt their operational modes according to available energy resources, actively shifting between processing-intensive and energy-conserving states. This biologically inspired approach ensures that devices operate maximally efficiently, analogous to biological systems' prioritising energy expenditure according to functional demand. Consequently, the AABCMS extends the bio-inspired framework beyond immunology into broader biological adaptive strategies, reinforcing the holistic bio-mimetic underpinnings of the system. Please see the table below for the conceptual diagram showing parallels between brain wave states and the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS). Both systems dynamically adapt energy expenditure according to operational demands, ensuring survival and functional longevity. In the neural system, different brainwave frequencies correspond to cognitive loads, while in the AABCMS, varying operational modes are triggered based on real-time battery health estimations.

Table 5: Conceptual diagram showing parallels between brain wave states and the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS).

Comparison between Brain Energy Adaptation and IoT Adaptive Amoeba Battery Curve Management	
Biological Neural System	**IoT Adaptive System (AABCMS)**
High Cognitive Load (Beta Waves, 13–30 Hz)	Active Processing Mode (ESP32 full power, secure communication)
Moderate Activity (Alpha Waves, 8–12 Hz)	Light Sleep Mode (moderate power conservation, faster wake-ups)
Low Activity / Drowsiness (Theta Waves, 4–7 Hz)	Deep Sleep Mode (aggressive power saving, longer wake intervals)
Deep Sleep (Delta Waves, 0.5–4 Hz)	ULP (Ultra-Low Power) Mode (battery preservation, minimal operation)

3.7 Bio-Inspired Communication Networks: Mycelium Systems

In addition to immune system paradigms and blockchain architectures, biological networks such as mycelium offer compelling models for adaptive, decentralised communication. Mycelium networks, formed by underground fungal hyphae, exhibit sophisticated resource distribution, threat detection, and environmental responsiveness without reliance on a centralised command structure. This decentralised sensing and signalling is facilitated through electrochemical gradients and molecular exchanges that parallel, in principle, how distributed IoT devices might coordinate authentication, threat detection, and energy conservation [82].

In recent studies, mycelium networks have been shown to exhibit patterns of electrical spiking behaviour reminiscent of neuronal or computational signalling [83]. Such properties suggest that future iterations of bio-inspired IoT frameworks could draw upon mycelial strategies for optimising communication efficiency under conditions of environmental uncertainty or infrastructural instability. The self-healing and redundancy features inherent in fungal networks could provide useful heuristics for designing resilient IoT systems operating in hostile or resource-constrained settings.

Thus, just as the Adaptive Amoeba Battery Curve Mapping Management System dynamically reallocates energy based on available resources, a mycelium-inspired model might further enhance system resilience through localised decision-making, distributed threat signalling, and adaptive load balancing. *Table 6* compares key features across biological analogies, underlining the broader potential of cross-disciplinary biomimicry in advancing the design of future intelligent systems.

Table 6: Comparative Features of Bio-Inspired Communication Systems

Comparison between Immune Systems, Blockchain, and Mycelium Networks for Bio-Inspired Communication			
Feature	**Immune System**	**Blockchain**	**Mycelium Network**
Architecture	Distributed, adaptive	Distributed, consensus-driven	Distributed, decentralised
Communication Method	Cytokine signalling, antigen presentation	Cryptographic validation, block propagation	Electrochemical signalling, nutrient transfer
Threat Detection	Foreign pathogen recognition	Transaction tampering detection	Environmental hazard sensing
Response to Threat	Rapid, targeted immune response	Forking or invalidating chains	Redirecting resources, avoiding threats
Memory Functionality	Immunological memory (e.g., memory T-cells)	Immutable ledger of historical transactions	Long-term memory via persistent hyphal structures
Fault Tolerance	High, redundant detection pathways	High, redundancy through multiple nodes	Extremely high, regenerative growth patterns
Energy Efficiency	Context-dependent (can be costly)	Moderate (depends on consensus mechanism)	Highly efficient resource distribution
Self-Healing	Replacement of damaged cells	Chain recovery via honest nodes	Regrowth of broken hyphae, rerouting signals

4.0 Chapter Four: Framework

4.1 Introduction

Chapter four proposes how the lightweight polymorphic security system would work and details each subroutine's various stages and methods, such as intrusion detection to detect any unapproved clients, a shared ledger for all approved clients, and the ability to change encryption keys and passwords. Additionally, this framework also introduces a novel adaptive amoeba battery curve mapping management system that any mobile IoT system can use.

A quick overview of the security system is as follows: the security system has a server that an approved client can connect to using the SSID of the server and a password. Once connected, the server will give the approved client a ledger shared between all approved clients. The communication between the server and the client is encrypted using a shared encryption key. This ledger shared a two-dimensional array containing new passwords to allow the client to connect to the server and new encryption keys to secure communication between client and server. If the server detects an unauthorised client trying to connect to the system, the server will send out a trigger to all approved clients. The trigger contains a number that corresponds to an element in the two-dimensional array. Therefore, all of the clients and the server change to the new password and a new encryption key, resulting in the unapproved client not knowing the new password or encryption keys.

Looking back at the previous chapter, the human immune system can detect foreign agents by using a known threat database, and if those known threats are detected, an immune response is activated. In cybersecurity, this is called a blocklist. In an IoT platform, the manufacturer or the person who is the system administrator would know the number of IoT clients that would be connected to the server. Therefore, instead of using a blocklist to reject any known incoming threats, using an allow list only to allow communication between the known IoT clients is a more straightforward approach. Hence, having an allow list approach covers how to detect any unapproved clients. However, suppose an unapproved client manages to get into the system by knowing the network's password. In that case, the trigger activates, allowing all approved clients to change to the new password and encryption key. This detection of unapproved clients (pathogens) is the immune response for this polymorphic security system.

The nature of the shared ledger comes from blockchain technology, which, in summary, is a growing list of records that is resistant to modification of its data and is distributed via a network through internal node communication. The ledger for this security system follows the same characteristics as the list of records, comprised of passwords and new encryption keys shared between all clients and the server.

Once all clients have the ledger, the clients will only read the ledger when the trigger is activated. The trigger is only activated when the server detects an unauthorised client is trying to connect to the network, meaning that the unauthorised client may have the initial password for the network. Since this unapproved client is not on the allow list, then the trigger is activated. The trigger is composed of an integer that relays the next element in the ledger array with new passwords and encryption keys. Henceforth, all clients receive the trigger and change to the new password and encryption key accordingly.

In this chapter, the framework visualisation for the polymorphic security system to better understand each function's input and output is explored and documented through various stages. These stages are as follows:

- Server and client model with a peer-to-peer node communication
- The approved client list for the allowed list
- Intrusion Detection
- Shared ledger
- Encrypted communication between client and server
- Synchronised reform from the ledger

Please see the following page for the Framework Overview. *Figure 4.1* below will be referenced throughout the rest of the chapter to explore the various stages stated above. Some functions are highlighted in blue in the figure, denoting that they are a part of the adaptive amoeba complexity. The blue functions indicate that these systems can increase or decrease complexity depending on the hardware used.

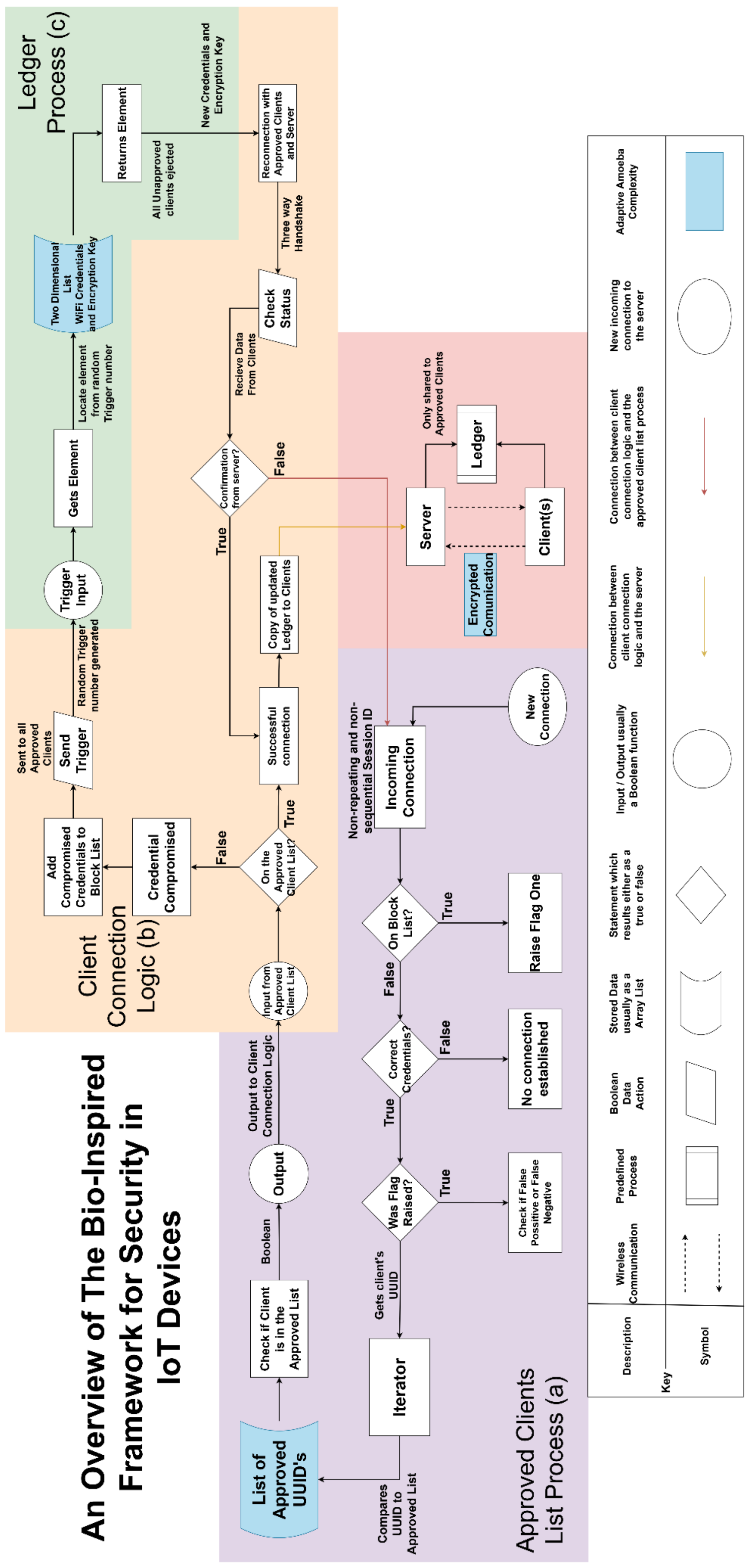

An Overview of The Bio-Inspired Framework for Security in IoT Devices
Approved Clients List Process (a)
Client Connection Logic (b)
Ledger Process (c)
List of Approved UUID's
Check if Client is in the Approved List
Boolean
Output
Output to Client Connection Logic
Iterator
Compares UUID to Approved List
Gets client's UUID
Was Flag Raised?
True
Check if False Possitive or False Negative
Correct Credentials?
True
False
No connection established
On Block List?
False
True
Raise Flag One
Incoming Connection
Non-repeating and non-sequential Session ID
New Connection
Input from Approved Client List
On the Approved Client List?
True
False
Credential Compromised
Add Compromised Credentials to Block List
Send Trigger
Sent to all Approved Clients
Random Trigger number generated
Trigger Input
Gets Element
Locate element from random Trigger number
Two Dimensional List WiFi Credentials and Encryption Key
Returns Element
All Unapproved clients ejected
New Credentials and Encryption Key
Reconnection with Approved Clients and Server
Three way Handshake
Check Status
Recieve Data From Clients
Confirmation from server?
True
False
Successful connection
Copy of updated Ledger to Clients
Server
Ledger
Only shared to Approved Clients
Client(s)
Encrypted Comunication
Key
Description
Symbol
Wireless Communication
Predefined Process
Boolean Data Action
Stored Data usually as a Array List
Statement which results either as a true or false
Input / Output usually a Boolean function
Connection between client connection logic and the server
Connection between client connection logic and the approved client list process
New incoming connection to the server
Adaptive Amoeba Complexity

Figure 4.1: Framework for the Bio-inspired Lightweight polymorphic security system for IoT devices

4.2 Approved Clients List Process (ACLP)

The beginning of the client journey starts with a new connection. The approved client list is the first line of defence against any unapproved clients. Additionally, the approved client list also acts as a RADIUS Server. The new client connection flows into the incoming connection function, where it is given a session ID that is non-repeating and non-sequential. Once given the session ID, the process then checks if the client has the correct credentials to connect to the server. The framework checks if the incoming client is on the block list and raises a flag if true. The purpose of the block list is to protect against denial-of-service attacks. If the credentials are incorrect, then there is no connection established between the incoming client and the server. If the credentials are valid, the server will check if the flag has been raised to see if there was a false positive or a false negative (the raised flag might occur due to failed resyncing). Then, it moves on to request the client's UUID to be compared using an iterator (a search function) with the approved UUID addresses. The iterator function will return and output either a "true" or "false" whether the UUID address is on the list or not. Referring back to the properties of adaptive immune responses, the ACLP deeply reflects features such as specificity, diversity, memory and specialisation.

4.3 Client Connection Logic (CCL)

The return output from the approved client list becomes an input variable to the client connection logic, as this function will react accordingly depending on the variable. If the variable is "true" (the connected client is on the UUID address list), this is considered a successful connection, and the server gives a copy of the ledger with the element location for the current password and encryption key. At this stage, the client and the server have successfully connected and can communicate with other clients and the server when necessary. Contrariwise, if the variable returns a "false", then the implication is that the current server credentials are compromised, and this unapproved client knows the password to connect. As soon as the "false" variable returns, the credentials used are added to the block list, and a trigger (composed from a random number generator) is sent to all approved connected clients. When a client receives the trigger value immediately, it will locate the ledger's element corresponding to the trigger value. The primary purpose of CCL is to reflect features of adaptive immunity, such as nonreactivity to self and contraction and homeostasis.

4.4 Ledger Process (LP)

The ledger, only shared between approved clients, contains a database comprised of a two-dimensional list, the password to connect to the server (Wi-Fi credentials) and the encryption key required to decipher the communication stream between client to client (node and peer-to-peer communication) and from client to server. The ledger's returned element allows the approved clients to reconnect with the new credentials to the server. As a result, all the unapproved clients get automatically rejected from the system as they do not have the newly updated credentials. When the clients attempt reconnection after the trigger is activated, a three-way handshake is performed, the server sends a synchronisation, the client syncs to the server synchronisation and acknowledges it, then the server acknowledges the client's acknowledgement. After a three-way handshake is successful, the client gets a copy of the updated ledger with the credentials' new element location. However, suppose the three-way handshake is unsuccessful (the server did not acknowledge the client). In that case, the client is reverted to an incoming connection to double-check if it is in the approved client list and if it is, it will get a copy of the ledger, and if it is not, it will be rejected. LP reflects adaptive immunity functions such as contraction and homeostasis and clonal expansion.

The server and client's communication stream is encrypted using a block cipher (or any encryption communication) to add additional security against data theft and intrusion, such as a man-in-the-middle attack or listening or probing intrusions.

4.5 The Adaptive Amoeba Complexity (AAC)

The Adaptive Amoeba Complexity (AAC) is a dynamic memory configuration system that changes the complexity of various elements of the framework to best suit the processing power of the hardware on which it is running. There are three main adaptive complexities within this framework; please see Figure 17 above, highlighted in blue. After a successful connection to the server is established, a copy of the ledger is given to the client, and the server will run a simple diagnostic check on the connected device to procure three specific information:

1. Processor Type and Bit Architecture, as this information can often be obtained from predefined macros or registers specific to each microcontroller family.
2. Memory Usage, which can be determined by reading specific registers or using system functions provided by the microcontroller's SDK.
3. Processing Power, which is more complex, but basic metrics like CPU usage can be estimated.

The result of these three pieces of information will directly reflect which encrypted communication protocol will be used, for example, one of the microprocessors that is connected might be an 8-bit processor that uses 35% of the memory (SRAM) and 40% of CPU, with this information it would be best to implement a lower process intensive encrypted communication stream to make sure the IoT device is not drawing too much current resulting in poorer battery life or a memory overflow error due to a large encryption key.

There are three implementations of the AAC in this framework: the first is Encrypted Communication, which dynamically changes according to the diagnostic check; the second is the ACLP, which is updated whenever new devices are added, or the block list is updated; and lastly, within the LP with the dynamic ledger for credentials and encryption keys. Depending on the network system topology, the AAC can be either set up universally between the server and client, or it can be individually set up between the broker and clients, as some brokers may have more connected clients or clients with more processing power than other brokers.

4.6 Adaptive Amoeba Battery Curve Mapping Management System

The primary purpose of the battery mapping system is to intelligently manage the power consumption of an IoT device by accurately predicting the battery's remaining capacity. This is achieved through continuous monitoring of the battery voltage, dynamic estimation of battery health, and adaptive adjustment of the device's operation modes to maximise battery life while maintaining functionality. By predicting battery percentage and adapting power usage based on real-time data, the system ensures that the IoT device can operate efficiently over extended periods, even in power-constrained environments.

In IoT systems (especially mobile), devices are often deployed in remote or difficult-to-access locations where frequent battery replacement or recharging is not feasible. In some cases, remote IoT devices are connected to solar panels or other forms of green energy to power the system, whereby power management and battery management are crucial to ensure maximum efficiency. These devices must operate efficiently for extended periods on limited power reserves.

Therefore, this Adaptive Amoeba Battery Curve Mapping Management System ensures enhanced energy efficiency by accurately predicting battery levels and dynamically adjusting the device's sleep modes as the system conserves energy, allowing the device to operate for longer periods without intervention. The system adjusts sleep intervals based on both battery percentage and health. Healthier batteries allow for shorter sleep intervals, enabling more frequent updates, while degraded batteries are managed more conservatively, extending their operational life. This helps conserve energy and extend the battery life, which is critical for IoT devices operating in the field for long durations without maintenance. The system ensures operational reliability as the device can continue functioning in low-power modes even as the battery depletes, preventing unexpected shutdowns and maintaining service continuity. This is particularly important in applications such as environmental monitoring, where consistent data collection is crucial. Lastly, the system also provides insights into the battery's discharge characteristics over time, enabling predictive and preventive maintenance. Operators can anticipate when a battery will need replacement or recharging, thus minimising downtime and ensuring continuous operation.

The novelty of this battery mapping system lies in its adaptive and predictive capabilities through data-driven adaptation, dynamic sleep management, and a circular buffer for data retention. Unlike traditional fixed mapping systems that use a static relationship between voltage and battery percentage, this system continuously collects data and refines its predictions over time. The system becomes more accurate and personalised to the specific battery's characteristics by storing and analysing past data. The system predicts battery life and uses that prediction to manage the device's operation modes dynamically. This is akin to a biological system where energy usage is optimised based on available resources, ensuring that critical functions are maintained even under constrained power conditions. The system employs a circular buffer to manage data storage in EEPROM, dynamically overwriting the oldest data points. This ensures that the most relevant data is retained while optimising memory usage and extending the lifespan of the EEPROM without the risk of memory overflow. This approach balances data retention with memory constraints, making it suitable for resource-limited IoT devices.

Initially, the battery mapping system operates using a predefined battery curve that estimates the relationship between voltage and battery percentage. This curve is based on typical battery discharge characteristics and provides a reasonable starting point for managing power. As the system operates, it collects real-world data points (voltage and corresponding estimated percentages). It refines its predictions through non-linear interpolation to more accurately predict battery percentage based on voltage readings. This approach reflects the non-linear discharge characteristics of batteries, offering a more precise estimation compared to linear models. The system estimates the battery's overall health and uses this information to inform power management decisions. This feature allows the system to trigger maintenance alerts when the battery is nearing the end of its life, ensuring timely interventions. This approach mimics biological systems, where organisms continuously gather information from their environment and adjust their behaviour to optimise survival and resource use.

The system begins by measuring the battery voltage through an ADC and applying an initial mapping function to estimate the battery percentage. This initial mapping acts as a baseline, similar to how an organism might assess its energy reserves at the start of a day. As the system collects more data over time, it shifts from relying on the initial mapping to using a more refined, non-linear interpolation model. This process mirrors how biological systems adapt and refine their energy management strategies based on experience and real-time conditions. The system dynamically adjusts its operation modes (e.g., light sleep, deep sleep, ultra-low power mode) based on the predicted battery percentage and health. This adaptive behaviour is analogous to how organisms modulate their activity levels in response to energy availability, conserving energy during times of scarcity while maintaining essential functions.

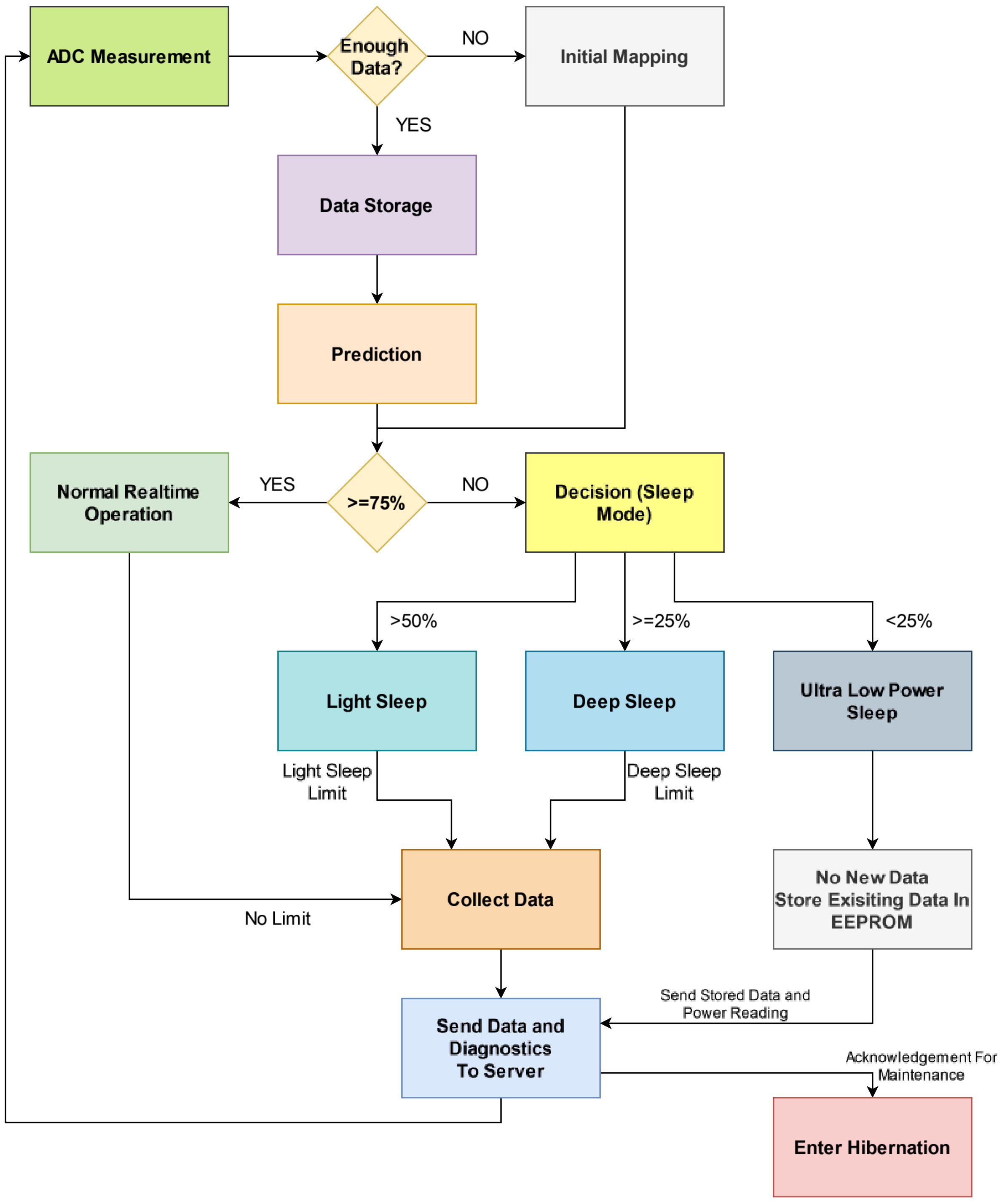


Figure 4.2: Block diagram for Adaptive Amoeba Battery Curve Mapping Management System

The block diagram for the Adaptive Amoeba Battery Curve Mapping Management System is shown in *Figure 4.2*. The battery management system begins with the ADC Measurement block, which reads the current battery voltage through the Analogue-to-Digital Converter (ADC). The ADC converts the analogue battery voltage into a digital value that the microcontroller can process. This voltage is a critical input for the subsequent steps in the system. Once the battery voltage is measured, the system evaluates whether sufficient historical data is stored to predict the battery percentage accurately. If enough data is available, the system stores the new measurement data. If not, the system moves to the Initial Mapping block.

In the Initial Mapping block, the system relies on a predefined voltage-to-percentage mapping to estimate the battery percentage, especially when there is insufficient data for more accurate predictions. This initial mapping provides a starting point for the battery status estimation until more refined data can be gathered. After completing the initial mapping, the system stores the estimated data and proceeds to the Prediction block.

The Data Storage block plays a crucial role in this system. The newly measured voltage and its corresponding battery percentage are stored in a circular buffer. This storage method ensures that the system retains the most recent data points without exceeding memory limits by replacing the oldest data when the buffer is full. With the data securely stored, the system moves to the Prediction block, which uses linear interpolation or other predictive methods to refine the battery percentage estimate based on the stored data. As the system gathers more data, this step becomes increasingly accurate.

After the prediction is made, the system checks whether the predicted battery percentage is greater than or equal to 75%. If the battery percentage is 75% or higher, the system continues with regular real-time operation, avoiding any sleep mode to maintain full functionality. However, if the battery percentage is below 75%, the system proceeds to the Decision (Sleep Mode) block, determining the appropriate sleep mode to conserve energy.

In the Decision (Sleep Mode) block, the system selects one of three sleep modes based on the predicted battery percentage: Light Sleep for battery levels between 50% and 75%, Deep Sleep for levels between 25% and 50%, and Ultra-Low Power Sleep (ULP) for levels below 25%. These sleep modes vary in the degree of power conservation and responsiveness, with ULP being the most energy-efficient but least responsive. After selecting the sleep mode, the system continues to the Collect Data block.

During the Collect Data phase, the system continues to gather and store data on battery voltage and other parameters, even while in sleep mode. This ongoing data collection is essential for refining future predictions and ensuring that the system remains accurate over time. Once the data is collected, it is sent to a remote server in the Send Data and Diagnostics to Server block. This transmission allows for remote monitoring and analysis, ensuring the device’s performance and battery health can be tracked over time.

In cases where the system is in ULP mode, it may stop collecting and storing new data to conserve power, instead relying on the data already stored in EEPROM. Maintenance might be required If the system has no recharging capabilities and the battery reaches a critically low level. The system sends all stored data, and the server sends an acknowledgement request for maintenance. If no maintenance is performed and the battery continues to deplete, the system will enter the Hibernation block. This final step ensures that the device remains protected and conserves as much power as possible until the battery can be recharged or replaced.

A key innovation of the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS) lies in its dynamic selection of encryption methods based on real-time battery availability. This mechanism, termed the Adaptive Encryption Engine, enables the system to adaptively shift between lightweight and more complex cryptographic schemes depending on energy constraints. The decision process is inspired by the metabolic prioritisation observed in biological organisms, where the immune system modulates its intensity based on physiological reserves.

When energy is scarce, only minimal response mechanisms are engaged; when energy is abundant, the full spectrum of immunological defence is deployed. In parallel, this system utilises the predicted battery percentage E_b to map to a corresponding encryption method M as follows:

Equation 4.1: Mapping for Encryption and Battery Percentages

$$M = f(E_b) = \begin{cases} XOR, & E_b \geq 15 \\ Caesar, & 15 \leq E_b < 30 \\ ROT13, & 30 \leq E_b < 50 \\ SPECK, & 50 \leq E_b < 75 \\ SIMON, & E_b \geq 75 \end{cases}$$

This mapping is directly implemented within the system firmware (see Appendix 16), enabling real-time responsiveness to energy states. Through this mechanism, the framework not only preserves battery life but also ensures an optimal balance between computational load and data security.

4.7 The Framework In Practice

The Bio-Inspired Lightweight Polymorphic Security System for IoT Devices is designed to manage connections and secure communications in an IoT environment dynamically. It mirrors biological immune system functions such as specificity, memory, and adaptability. It ensures that only approved clients can communicate with the server while dynamically adjusting its behaviour based on the capabilities of the hardware in use.

The system is divided into several key components:

- Approved Clients List Process (ACLP): This is the first line of defence. It checks if a connecting client is approved by comparing its UUID against a list of approved UUIDs. If the client is not approved, the connection is denied, and the credentials are added to a block list if necessary.
- Client Connection Logic (CCL): This logic handles the connection of approved clients. It ensures that only clients with the correct credentials and UUIDs can connect and receive the necessary ledger information for secure communication.
- Ledger Process (LP): The ledger is a dynamically updated list shared among approved clients. It contains the credentials and encryption keys required for secure communication. The ledger is updated and shared only with approved clients.
- Adaptive Amoeba Complexity (AAC): This component dynamically adjusts the encryption protocols and other system parameters based on the hardware capabilities of the connected devices, ensuring that the system remains efficient and secure regardless of the device's processing power.

Depending on the use case scenario, an IoT network system comes in various shapes, sizes and power requirements. Typically, an IoT network will always have a client responsible for some form of environmental measurement or interaction. The client collects data and either temporarily stores it locally for edge processing or uploads it to a server for long-term storage. In most applications, the client collects sensor data and then directly sends it to the server for further processing for end-user applications. See *Figure 4.3*.

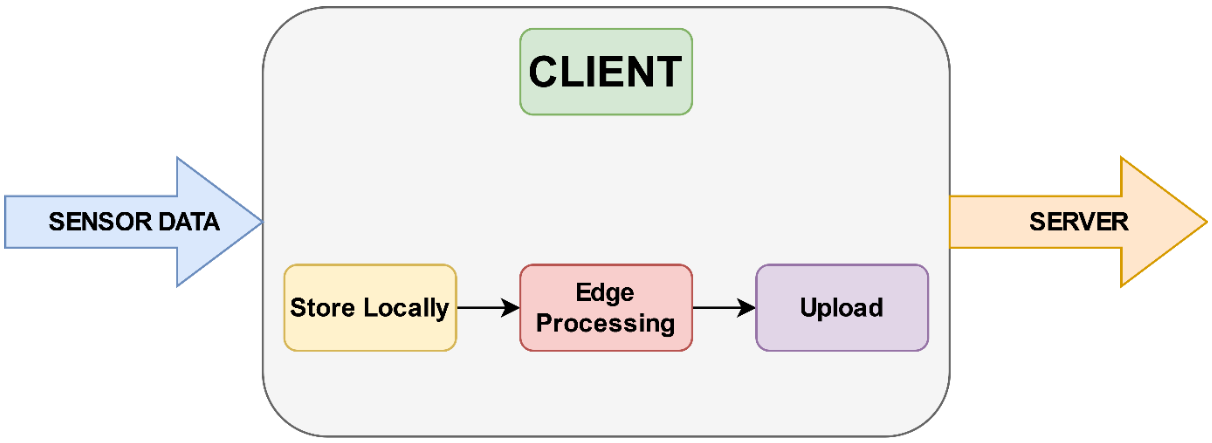


Figure 4.3: Generic IoT Client Responsibilities

The server is responsible for long-term storing, processing data and maintaining client updates. In smaller topologies, the server receives the data from the client to process and display the information. Refer to *Figure 4.4*. In more extensive topologies and applications, it is common for the server also to have a broker (or router) that manages the clients. An example of this is when there are more clients than the server can handle directly, as it could unintentionally cause a Denial of Service due to the amount of data the server has to process. Therefore, having a broker alleviates the server's processing and distributes it. An alternative is to have a distributed system which has both responsibilities for the client and the server.

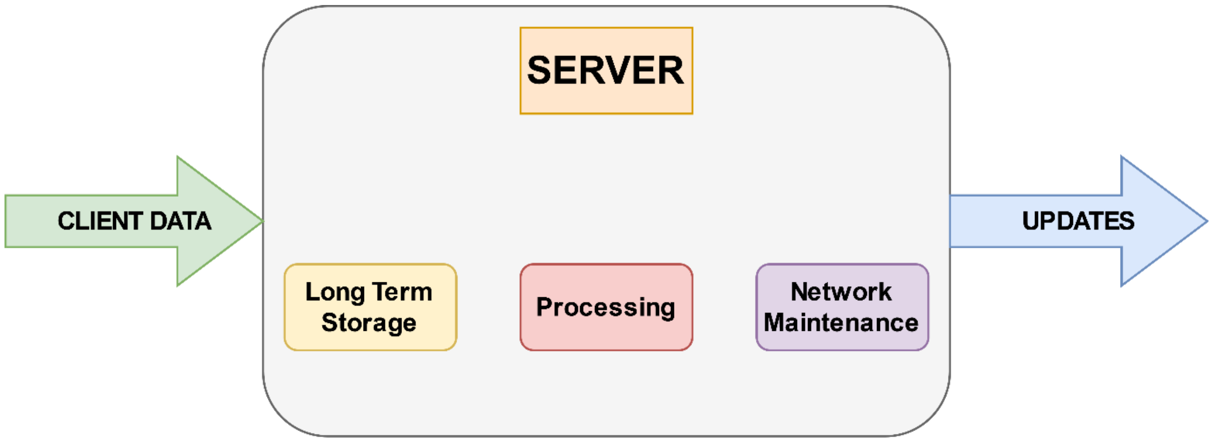


Figure 4.4: Generic IoT Server Responsibilities

The Bio-inspired Lightweight Polymorphic Security System Framework is best described as a multi-layered cell wall that performs various checks before allowing a new connection to enter. To help visualise this process, the following figures describe the framework as a new client connection.

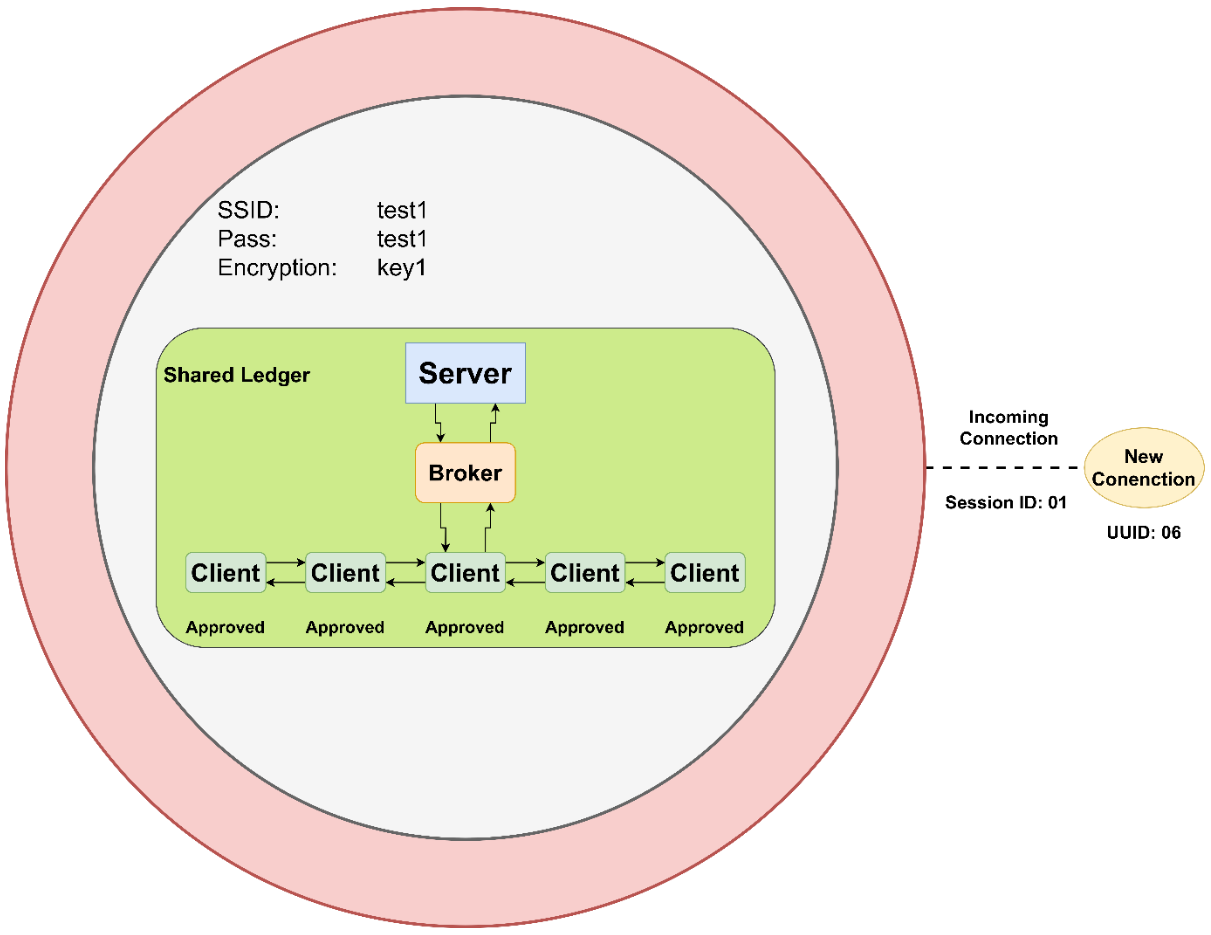


Figure 4.5: Visualisation of Incoming new connection

In *Figure 4.5*, a new connection occurs, requesting the Universally Unique Identifier (UUID) and creating a non-repeating and non-sequential session ID within this process. In this figure, the cell wall contains the server, broker, clients, and the shared ledger, which contains the SSID and password credentials with the encryption key. Currently, the SSID and password are set to "test1" while the encryption key is set to "key1".

Implementing non-repeating and non-sequential session IDs significantly enhances the security and integrity of web applications. These session IDs are crucial in preventing session hijacking and session fixation attacks by making it difficult for attackers to guess or infer valid session IDs through brute force or other methods. Additionally, they obscure patterns that attackers could exploit, thus improving confidentiality. The uniqueness of non-repeating session IDs avoids collisions, ensuring each session is distinct and secure.

This practice also aligns with regulatory and compliance standards such as OWASP and GDPR, reducing the risk of penalties and enhancing the organisation's security posture. Furthermore, non-repeating session IDs offer protection against replay attacks by preventing the reuse of session IDs. To effectively implement these IDs, cryptographically secure random number generators are used (see the implementation chapter), sufficient length and complexity are ensured, session IDs are securely stored and transmitted, and periodic rotation is considered.

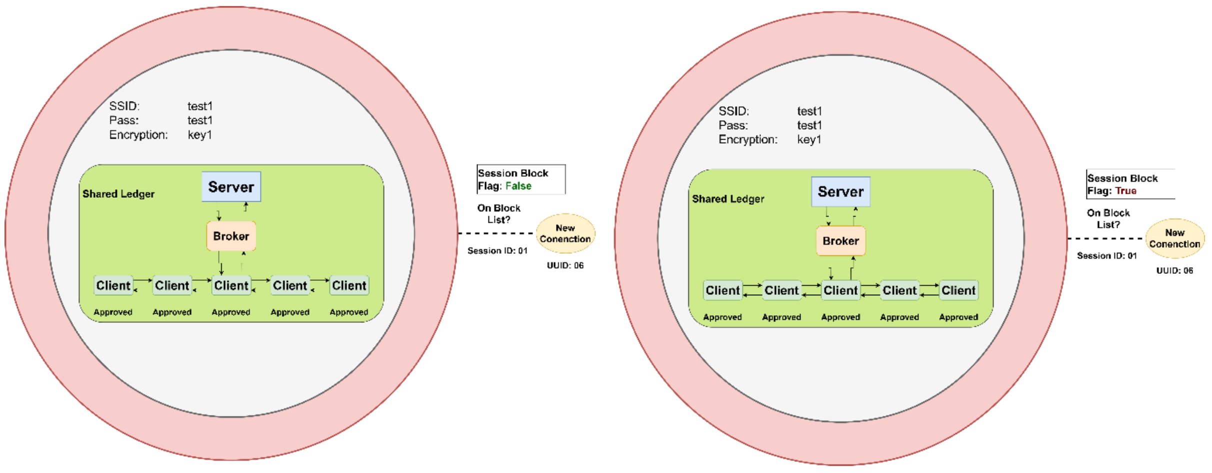


Figure 4.6: Visualisation of session block list check

In *Figure 4.6*, the framework checks if the new connection's session ID is on the block list; if not, it can continue to the next stage, and the flag remains false; if the session ID is on the block list, then the flag is set to true. During the testing stages (see the testing chapter), some connections were flagged up not because of bad actors but because of resyncing multiple times; due to this, there was extra implementation to check if the flag was raised and then to confirm a false positive or a false negative.

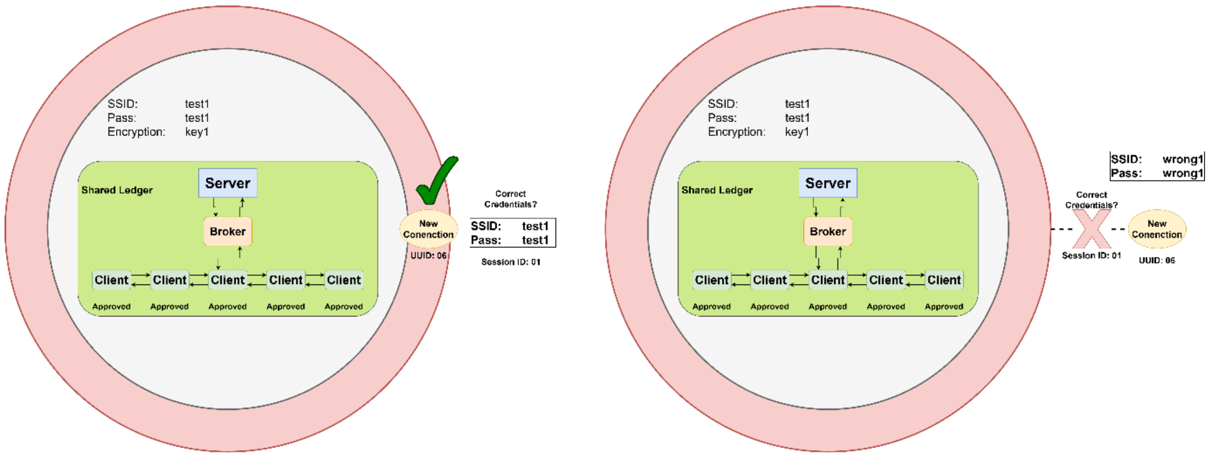


Figure 4.7: Visualisation of the credentials check

In *Figure 4.7*, the SSID and password credentials are checked. This stage is a precursor to see if there are any compromised credentials. If the new connection is successful, it will then move into the next stage; if the credentials are incorrect, then the system will disconnect from the new connection.

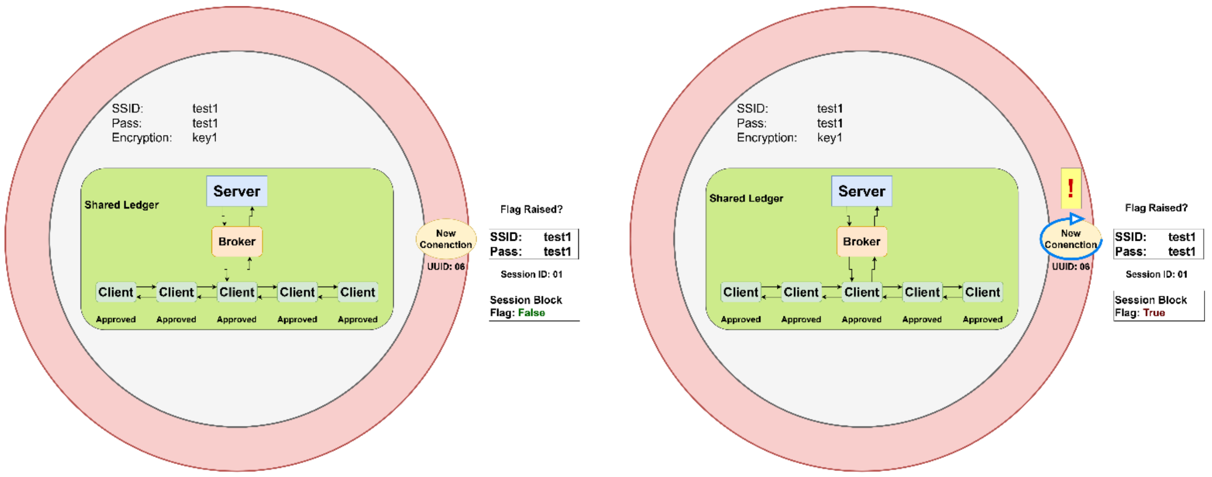


Figure 4.8: Visualisation of the flag checker

Figure 4.8 examines whether the flag has been raised. If the credentials are validated, the server verifies the flag status to determine the occurrence of false positives or false negatives, which may arise due to failed synchronisation attempts. In the case of a false positive, the session block list is updated, allowing the new connection to retry the process. Conversely, if a false negative is detected and the new connection fails the credentials check, the connection is rejected. However, if a false negative is detected and the credentials check is passed, the approved UUID triggers the immune response.

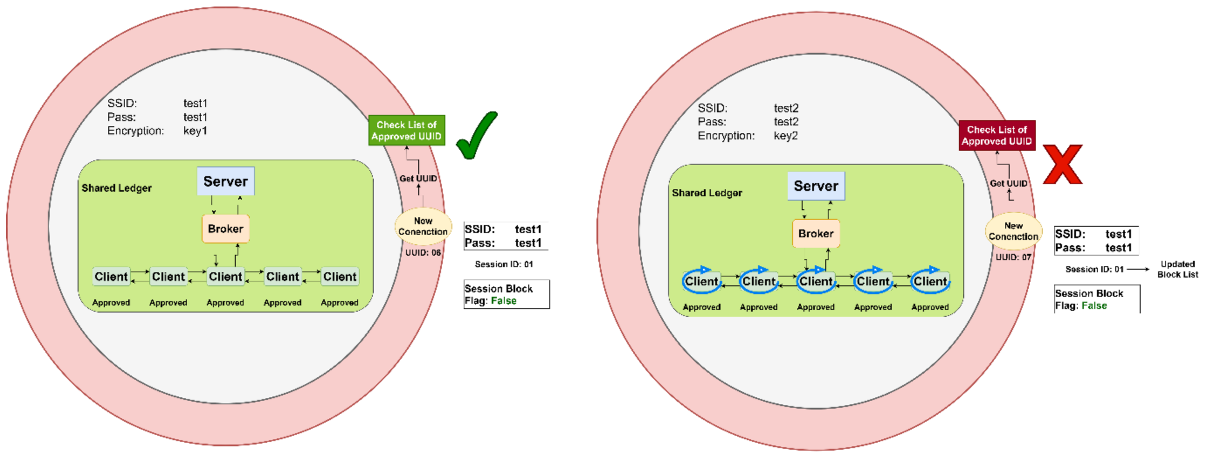


Figure 4.9: Visualisation of the Approved Client (UUID) List

Figure 4.9 illustrates the process of verifying the UUID against the approved client list. The new connection must provide the UUID, which is then compared to the approved client list, yielding a boolean result. If the result is true, the client is permitted to pass through the cell wall and register as a new client. Conversely, if the result is false, the immune response is activated, indicating that the new connection has bypassed the block list and successfully provided valid credentials, suggesting that the credentials have been compromised. Upon raising the flag, the session ID and the compromised credentials are updated in the block list and the ledger. Subsequently, a trigger is sent to the shared ledger, prompting all approved clients to adopt the new credentials and encryption key, necessitating their reconnection. During this process, the unauthorised new connection is rejected from the system and automatically added to the session block list.

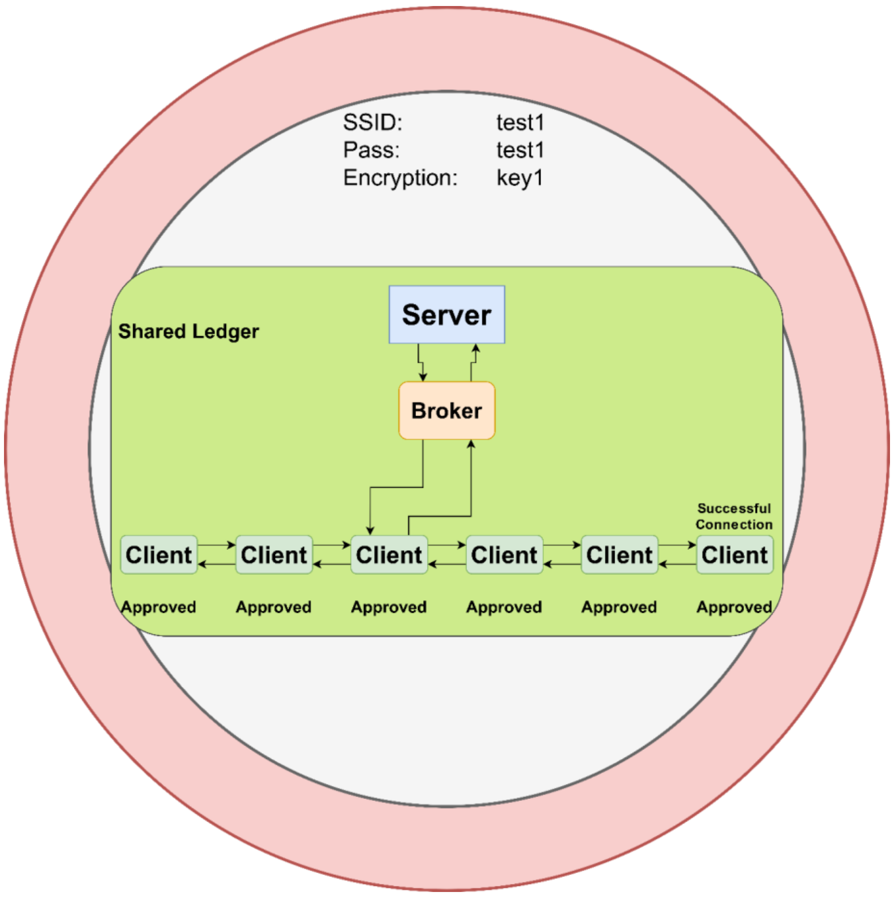


Figure 4.10: Visualisation of a successful integration of a new connection

Figure 4.10 illustrates the successful integration of a new connection following approval from the authorised client list. Upon acceptance, depending on the network topology employed, the server or other clients will disseminate the shared ledger to the new client. It is important to note that the SSID, password, and encryption key remain unchanged, as no security breach or compromise has been detected or flagged. This process can be adapted to alter the SSID, password, and encryption keys either after each new connection or after a predetermined number of connections have been established with a broker (or server, depending on the topology).

The Bio-Inspired Lightweight Polymorphic Security System for IoT devices dynamically manages connections and ensures secure network communication. The system begins with the Approved Clients List Process (ACLP), where incoming connections are checked against a list of approved UUIDs. If the client is approved, the connection is allowed, and the system proceeds to the Client Connection Logic (CCL). Here, the client is provided with the necessary credentials and encryption keys stored in the ledger, allowing for secure communication. The ledger itself is managed by the Ledger Process (LP), which updates and distributes the ledger to all approved clients as needed. Finally, the Adaptive Amoeba Complexity (AAC) ensures that the system's complexity and encryption protocols are dynamically adjusted based on the hardware capabilities of the connected devices, ensuring optimal performance and security.

This system reflects the adaptive nature of biological immune systems, with features such as specificity, memory, and adaptability. The system is designed to be resilient, secure, and efficient, making it well-suited for use in IoT environments where resource constraints and security are paramount.

5.0 Chapter Five: From Theory to Practice (Implementation)

5.1 Introduction

This chapter aims to explore the logical representations and workings from the previous framework chapter to develop mathematical formulas and the corresponding programming logic needed for implementation. Every implementation will be written in C due to its programming efficiency, and the hardware used is a custom development board made with an ESP32-S3 processor, which can be found in Appendix 2.

The system comprises an IoT client-server architecture designed for secure, efficient, and adaptive communication between IoT devices. The server manages multiple client sessions, ensures secure communication through encryption, and adapts to varying battery levels of IoT devices. The key components of the system include secure session management, adaptive power management, and encrypted data transmission, all integrated into a shared ledger for synchronisation.

5.2 Basic Client and Server on IoT Devices

The server-client system implemented on the custom development board made with an ESP32-S3 chip is designed to enable wireless communication between two devices over a Wi-Fi network. The server is responsible for listening to incoming connections, processing client requests, and sending appropriate responses. Conversely, the client initiates a connection to the server, sends a request, and then processes the server's response. This system is typical in IoT applications where multiple devices need to communicate with one another over a local network or the internet. Please see the complete code attached in Appendix 3, as it will be continuously referenced throughout this sub-chapter.

The server-client system described here involves two ESP32-S3 chips: one configured as a server and the other as a client. Typically, the server would be hosted on dedicated hardware, but for this implementation, it is essential to see how this system works with limited resources. This system allows the client to connect to the server over a Wi-Fi network, exchange data, and receive a response. The communication protocol used here is based on TCP/IP, specifically HTTP over port 80, a standard setup for lightweight web servers on IoT devices.

On the server-side operation, the WiFi connection is initialised as the server first connects to a Wi-Fi network using the provided SSID and password:

Equation 5.1: Server WiFi Initialisation

$$Server_{IP} = WiFi.begin(SSID, Password)$$

The server joins the network and is assigned an IP address ("*Server_IP*"). This IP address is crucial as clients will use it to connect to the server. Once connected to the Wi-Fi, the server initialises a Wi-Fi server object on port 80, which is the default port for HTTP communication:

Equation 5.2: Server WiFi Port

$$Server = WiFiServer(Port)$$

The server then begins listening for incoming client connections:

Equation 5.3: Server Initialisation

$$Server.begin()$$

To listen for client connections, the server enters a loop that continuously checks for incoming client connections. If a client connects, the server establishes a communication channel with the client:

Equation 5.4: Server Checking for Client Connections

$$Client_{Socket} = Server.available()$$

This function checks if a client is available and returns a "*WiFiClient*" object representing the client. Upon connection, the server reads the incoming data from the client line by line:

Equation 5.5: Server Reading Incoming Client Data

$$Data = Client.read()$$

The server processes the data, and if it detects an HTTP request (indicated by a blank line), it sends an HTTP response back to the client:

Equation 5.6: Sending HTTP Response from Server to Client

$$Client.send(HTTP_{Response})$$

The response includes a simple HTML page, which the client will display if it is capable of rendering HTML. After sending the response, the server allows the client some time to receive the data and then closes the connection. This ensures that resources are freed and the server is ready to handle new connections:

Equation 5.7: Closing Client Connections

$$Client.stop()$$

On the client side, the client also begins by connecting to the same Wi-Fi network as the server:

Equation 5.8: Client WiFi Initialisation

$$Client_{IP} = WiFi.begin(SSID, Password)$$

The client is similarly assigned an IP address (“*Client_IP*”), although this is typically not directly used in the communication process. The client attempts to establish a connection with the server using the server's IP address (Server_IP) and port 80:

Equation 5.9: Client Connecting with Server IP and Port

$$Connection = Client.connect(Server_{IP}, Port)$$

If the connection is successful, the client is ready to send and receive data. The client sends a string of data to the server, which could represent a request or some other information:

Equation 5.10: Client Sending Data to Server

$$Client.send(Data)$$

In this example, the string "The quick brown fox jumps over the lazy dog 1234567890!@#$%^&*()_+-=[]{}|;':,.<>/?\n\t" is sent to the server as it includes all alphanumeric characters with a new line and tab blanked space. The client then waits for a response from the server. The response is read line by line:

Equation 5.11: Client Awaiting for Server Response

$$Response = Client.readStringUntil(\text{"}^{\circ})$$

The client processes and displays this response, which, in this case, is an HTML page sent by the server. Do note that the carriage return “\r” can also be denoted as 0xD in Hex; the purpose is to return to the beginning of the current line without advancing downward. After receiving the response, the client closes the connection to free up resources. The client is programmed to periodically reconnect to the server every 10 seconds, allowing it to send new requests or data and receive updated responses. This ensures ongoing communication between the client and the server, which can be helpful in applications requiring regular updates.

Overall, the server-client system can be represented mathematically by the following sequence of events:

Equation 5.12: Server IP Assignment

$$Server\_IP = WiFi.begin(SSID, Password)$$

Equation 5.13: Client IP Assignment

$$Client\_IP = WiFi.begin(SSID, Password)$$

Equation 5.14: Server Port Listening

$$Socket = Server.listen(Port)$$

Equation 5.15 Client Connection Attempt

$$Connection = Client.connect(Server\_IP, Port)$$

Equation 5.16: Data Transmission (Client to Server)

$$Client.send(Data)$$

Equation 5.17: Data Reception and Response (Server to Client)

$$Response = f(Data)$$

$$Client.read(Response)$$

Equation 5.18: Connection Termination

$$Client.stop()$$

The server-client system on the ESP32-S3 chip operates through a well-defined sequence of connection, communication, and disconnection. The server initialises and continuously listens for incoming connections while the client attempts to connect, sends data, and processes the server's responses. This system allows real-time data exchange over a Wi-Fi network, making it suitable for various IoT applications such as remote monitoring, control systems, and simple web-based interfaces. The mathematical expressions captured the communication flow's essence, illustrating the process from both the server's and client's perspectives. The periodic reconnection mechanism implemented in the client ensures that communication remains active and up-to-date, which is crucial for maintaining the functionality and reliability of the system in a dynamic environment.

5.3 Adaptive Encryption Methods (Adaptive Amoeba Complexity)

The adaptive encryption system is designed to dynamically choose between different encryption algorithms and configurations based on the available memory and the specific hardware platform detected. Depending on the resources at hand, the system can deploy both lightweight and more complex encryption schemes, ensuring optimal security and performance on the ESP32-S3 and other microcontroller architectures. The system leverages well-known cryptographic algorithms, including XOR encryption, Caesar cipher, ROT13, and the more sophisticated SPECK and SIMON block ciphers. For the complete documentation of the code, please refer to Appendix 3.

Upon system initialisation and hardware detection, the system first measures the available memory on the microcontroller:

Equation 5.19: Measuring Available Memory

$$Available\ Memory = freeMemory()$$

The system then identifies the specific microcontroller architecture (e.g., ESP32, STM32, Arduino AVR, etc.). This detection allows the system to make informed decisions about which encryption algorithms to deploy based on the capabilities and constraints of the hardware. For example, if an ESP32-S3 chip is detected with more than 10,000 bytes of free memory, the system will opt for more robust encryption schemes, such as the 256-bit configurations of SPECK and SIMON. If less memory is available, the system will downgrade to 128-bit versions of these algorithms or choose more straightforward encryption methods, such as XOR or ROT13.

The system supports several encryption algorithms, each with specific computational requirements and security properties:

Each XOR Encryption operation byte of the input data is XORed with a constant key (0xAA). This is a simple yet fast method of encryption that requires minimal computational resources. The XOR operation is repeated for each byte in the data string:

Equation 5.20: Encryption for XOR with Constant Key

$$Encrypted\ Byte = Data\ Byte \oplus 0xAA$$

Caesar cipher is a classical encryption method where each alphabetical character in the input data is shifted by a fixed number of positions (in this case, three positions). This method is suitable for environments with minimal resources:

Equation 5.21: Encryption for Caesar with a Fixed Number of Positions

$$Encrypted\ Character = (Character - Offset + 3) mod26 + Offset$$

ROT13 is a specific case of the Caesar cipher, where the shift is fixed at 13 positions. This is a lightweight, symmetric encryption technique where applying the algorithm twice returns the original text:

Equation 5.22: Encryption for ROT13

$$Encrypted\ Character = (Character - Offset + 13) mod26 + Offset$$

SPECK and SIMON Block ciphers are modern lightweight block ciphers designed for constrained environments. SPECK and SIMON perform a series of rounds (22 for SPECK-128 and 32 for SIMON-128, with more rounds for the 256-bit versions) of bitwise operations, additions, and rotations:

Equation 5.23: SPECK and SIMON General Equation for Bitwise Operations, Additions, and Rotations

$$x = (x\ >>\ 8) \oplus k, y = (y\ <<\ 3) \oplus x$$

Here, x and y are the halves of the plaintext block, and k is the subkey for that round. The number of rounds r is determined by the key size, with larger keys requiring more rounds to achieve greater security.

Equation 5.24: For Each Round In SPECK

$$x_i = (x_i - 1 >> 8) + y_i - 1 \oplus k_i, y_i = (y_i - 1 << 3) \oplus x_i$$

Equation 5.25: For Each Round In SIMON

$$y_i = (y_i - 1 >> 1) \oplus x_i - 1, x_i = (x_i - 1 << 1) \oplus k_i$$

The system's adaptability is rooted in its ability to choose the most appropriate encryption algorithm based on the current memory availability. If sufficient memory is available, the system opts for the more robust 256-bit versions of SPECK or SIMON, which provide a higher level of security due to increased key size and number of rounds. In contrast, if memory is constrained, the system selects the 128-bit versions or even falls back to simpler algorithms like XOR, Caesar cipher, or ROT13, which are less secure but more efficient.

The choice of encryption method can be expressed as a function of available memory M:

Equation 5.26: Choice of Encryption Method Based on Available Memory

$$Encryption\ Method = \begin{Bmatrix} SPECK256\ \&\ SIMON256 & if\ M > 10000\ bytes \\ SPECK128\ \&\ SIMON128 & if\ 5000 < M\ \leq 10000\ bytes \\ XOR, Caesar, ROT13 & if\ M \leq 5000\ bytes \end{Bmatrix}$$

This decision process ensures that the system remains operational and secure regardless of the specific constraints imposed by the hardware. The system monitors the time taken to perform each encryption method by capturing the start and end time in microseconds:

Equation 5.27: Monitoring Time Taken for Each Encryption

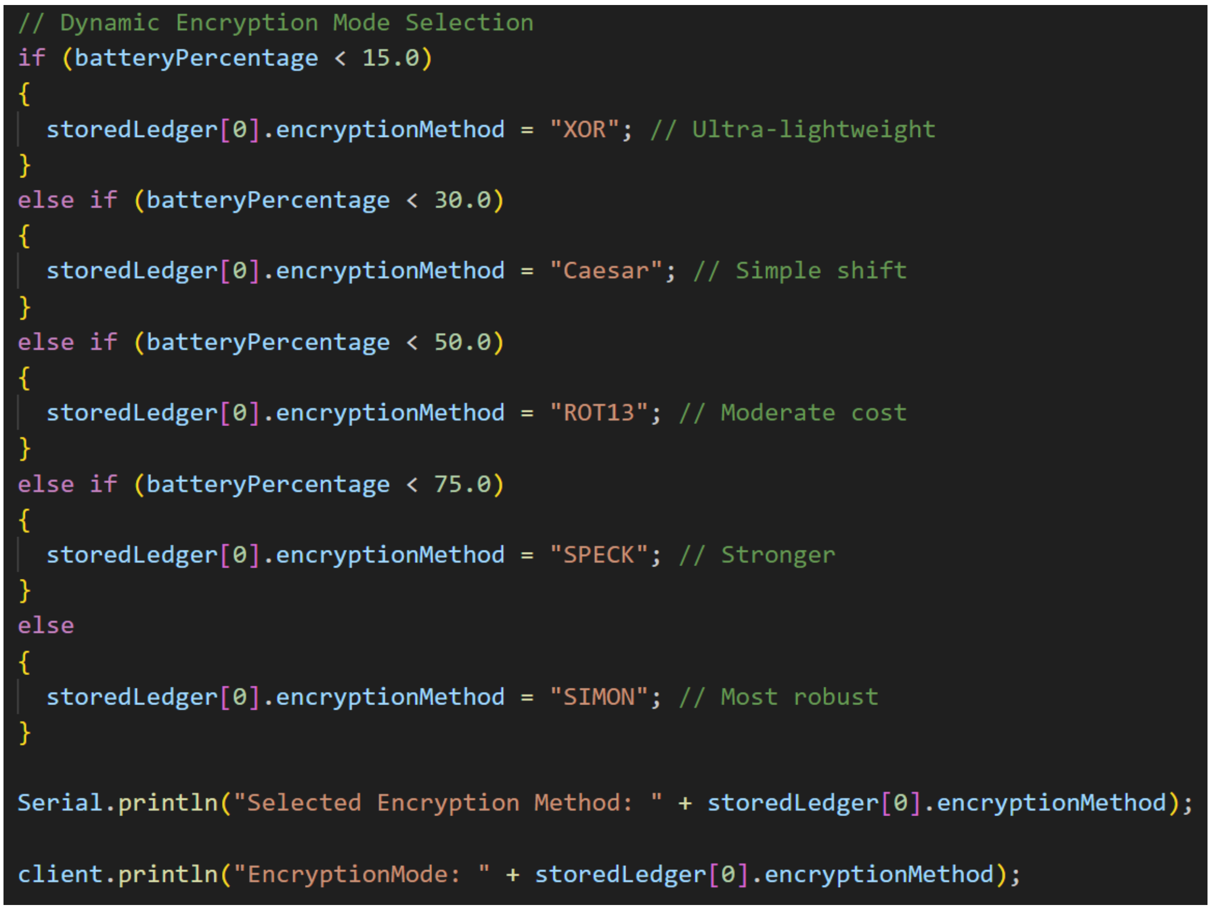
$$Time\ Taken = end_{time} - start_{time}$$

This measurement allows the system to evaluate the efficiency of each encryption algorithm in real time, providing insights into the trade-offs between security and performance. The timing information can be crucial for applications where encryption strength and speed are critical factors.

The adaptive encryption system on the ESP32-S3 chip (which can be applied to any processor) exemplifies a flexible and efficient approach to securing data in resource-constrained environments. The system balances security needs with computational efficiency by dynamically selecting the encryption method based on available memory and hardware capabilities. This adaptability makes it particularly well-suited for IoT devices, where resources are often limited, and security requirements vary widely depending on the application. The mathematical expressions used to describe the encryption processes provide a clear framework for understanding how the system operates, ensuring that it can be optimised and extended as needed.

Building upon the theoretical model described in Chapter 4, the firmware implementation integrates an if-else cascade that evaluates the current battery percentage and dynamically assigns the encryption method to be used in the outgoing transmission process. This method assignment is stored within $storedLedger[0].encryptionMethod$ and subsequently executed during the $encryptDecryptData()$ call. This approach ensures seamless runtime adaptability with minimal computational overhead.

For example, when batteryPercentage falls below 15%, the system automatically defaults to XOR encryption. As battery reserves increase, progressively more complex methods, Caesar, ROT13, SPECK, and finally SIMON, are employed. This design aligns with the Amoeba-inspired adaptability principle of the system, dynamically morphing cryptographic behaviour to align with physiological (electrical) capacity. Please refer to *Figure 5.1*.



```
// Dynamic Encryption Mode Selection
if (batteryPercentage < 15.0)
{
  storedLedger[0].encryptionMethod = "XOR"; // Ultra-lightweight
}
else if (batteryPercentage < 30.0)
{
  storedLedger[0].encryptionMethod = "Caesar"; // Simple shift
}
else if (batteryPercentage < 50.0)
{
  storedLedger[0].encryptionMethod = "ROT13"; // Moderate cost
}
else if (batteryPercentage < 75.0)
{
  storedLedger[0].encryptionMethod = "SPECK"; // Stronger
}
else
{
  storedLedger[0].encryptionMethod = "SIMON"; // Most robust
}

Serial.println("Selected Encryption Method: " + storedLedger[0].encryptionMethod);

client.println("EncryptionMode: " + storedLedger[0].encryptionMethod);
```

Figure 5.1: Dynamic Encryption Mode Selection Embedded and Automated within the Adaptive Amoeba Battery Curve Mapping Management System in C

5.4 Encryption between Client and Server Communication Streams

Lightweight encryption is a critical component in resource-constrained environments such as IoT devices. It aims to provide adequate security while minimising computational overhead and power consumption. The ESP32-S3 chip, being a powerful yet constrained microcontroller, benefits from such approaches, especially when deployed in a client-server architecture. The goal here is to encrypt and decrypt data transmitted between the client and server using lightweight encryption techniques, ensuring data confidentiality without significant performance penalties.

On the client side, the operation is split into three stages: the preparation and encryption of the data, the transmission to the server, and the receiving and decrypting of the server response. During the data preparation and encryption stage, the client prepares the data it intends to send to the server. This data could be a simple message or sensor readings from an IoT device. A lightweight encryption algorithm, such as XOR encryption or a simple Caesar cipher, is applied to the data:

Equation 5.28: General Encryption with Data and Key Variables

$$EncryptedData = Encrypt(Data, Key)$$

For XOR encryption, each byte of the data is XORed with a key:

Equation 5.29: General XOR Encryption with Data and Key Variables

$$EncryptedByte_i = Data_i \oplus Key$$

For a Caesar cipher, each character in the data is shifted by a fixed number of positions in the alphabet.

The second stage is the transmission to the server, as the encrypted data is then transmitted to the server over a TCP/IP connection:

Equation 5.30: Client Sending Encrypted Data to Server Over TCP/IP

$$Client.send(Encrypted\_Data)$$

Lastly, in the receiving and decrypting server response, the client waits for a response from the server, which is typically also encrypted. Upon receiving the encrypted response, the client decrypts it using the same algorithm and key used for encryption:

Equation 5.31: Server Decrypting Response From Client

$$Decrypted\_Response = Decrypt(Encrypted\_Response, Key)$$

The server-side operations are as follows: receiving and decrypting client data, processing data and generating a response, and sending the encrypted response. The server receives the encrypted data from the client:

Equation 5.32: Receiving Encrypted Data from Client

$$Encrypted\_Data = Server.receive()$$

The server then decrypts this data using the corresponding lightweight decryption algorithm:

Equation 5.33: Lightweight Decrypting Received Data from Client

$$Decrypted\_Data = Decrypt(Encrypted\_Data, Key)$$

After decrypting the data, the server processes it as needed. This could involve logging the data, triggering a control action, or generating a response. The server then encrypts the response using the same lightweight encryption algorithm:

Equation 5.34: Server Generating Lightweight Encrypted Response

$$Encrypted\_Response = Encrypt(Response, Key)$$

Lastly, the encrypted response is sent back to the client:

Equation 5.35: Server Sending Encrypted Response

$$Server.send(Encrypted\_Response)$$

Block cipher encryption provides a higher level of security compared to lightweight encryption techniques. It operates on fixed-size blocks of data and uses complex key schedules and multiple rounds of encryption to ensure data security. In this system, algorithms like SPECK or SIMON may be used, which are designed for resource-constrained environments but still provide robust encryption.

For both lightweight and block ciphers, the client and server operations remain the same as the client side. The operation is split into three stages: the preparation and encryption of the data, the transmission to the server, and the receiving and decrypting of the server response. The client prepares the data and breaks it into blocks of a fixed size (e.g., 64 bits or 128 bits). Each block of data is encrypted using a block cipher algorithm such as SPECK or SIMON:

Equation 5.36: Client Preparation for Encryption Block Size

$$Ciphertext_i = Encrypt\_Block(Plaintext_i, Key\_Schedule)$$

The encryption process involves multiple rounds of operations (e.g., XOR, rotation, and addition) with subkeys derived from the main key.

At the second stage, the encrypted blocks are concatenated and sent to the server as a single encrypted message:

Equation 5.37: Client Concatenating Encrypted Blocks

$$Client.send(Encrypted\_Message)$$

Lastly, the client receives the encrypted response from the server, which is also structured in blocks. Each block is decrypted using the corresponding decryption process:

Equation 5.38: Client Decrying Blocks into Paintext

$$Plaintext_i = Decrypt\_Block(Ciphertext_i, Key\_Schedule)$$

The server-side operations are as follows: receiving and decrypting client data, processing data and generating a response, and sending the encrypted response. The server receives the encrypted message from the client, which is composed of multiple encrypted blocks:

Equation 5.39: Server Receiving Multiple Encrypted Blocks

$$Encrypted\_Message = Server.receive()$$

The server decrypts each block using the block cipher decryption process:

Equation 5.40: Server Decrypting Multiple Encrypted Blocks

$$Plaintext_i = Decrypt\_Block(Ciphertext_i, Key\_Schedule)$$

Secondly, the server processes the decrypted data and prepares a response. This response is then encrypted in blocks using the block cipher encryption:

Equation 5.41: Server Preparing and Encrypting Response

$$Ciphertext_i = Encrypt\_Block(Plaintext_i, Key\_Schedule)$$

Lastly, the encrypted response blocks are concatenated and sent back to the client:

Equation 5.42: Server Sending Encrypted Response

$$Server.send(Encrypted\_Response)$$

The mathematical foundation lies in the operations applied to the data during encryption and decryption in both the lightweight and block cipher encryption systems. The lightweight encryption methods, such as XOR or Caesar cipher, involve simple arithmetic operations that can be expressed as:

Equation 5.43: General Arithmetic Operations for XOR and Caesar

$$Encrypted\_Byte = Data\_Byte \oplus Key(XOR)$$

$$Encrypted\_Character = (Character - Offset + Shift) mod\ 26 + Offset(Caesar\ Cipher)$$

For block ciphers like SPECK or SIMON, the encryption of each block can be represented as:

Equation 5.44: General Block Representation for SPECK and SIMON

$$Ciphertext_i = Encrypt\_Block(Plaintext_i, Key\_Schedule)$$

Where the encryption block function involves multiple rounds of operations, including bitwise shifts, rotations, and modular addition:

Equation 5.45: Block Functions with Multiple Rounds of Operations

$$x_i = (x_i - 1 >> 8) + y_i - 1 \oplus k_i$$

$$y_i = (y_i - 1 << 3) \oplus x_i$$

Given the appropriate key, these operations ensure that the ciphertext is a highly secure and non-reversible transformation of the plaintext.

The adaptive encryption system implemented on the ESP32-S3 chip provides a flexible and secure communication method between a client and server. By selecting the appropriate encryption method that is lightweight for resource-constrained scenarios and blocks ciphers for more secure communication, the system balances security and performance according to the available resources and the application's needs. The mathematical expressions involved in the encryption and decryption processes highlight the system's ability to securely handle data in various contexts, ensuring confidentiality and integrity in client-server communications.

5.5 Reading and Sending Sensor Data via Encryption Communication Streams

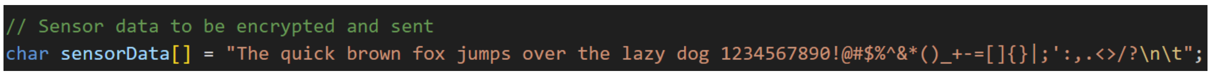
Before exploring how the encrypted communication streams work, there needs to be an understanding of how the system handles different scenarios. When a new client connects to the server for the first time, the following sequence of operations occurs: Session ID Generation initiates as the server generates a new session ID and sends it to the client. The client sends identification to the server, including its UUID, session ID, and processing power information. Server Verification occurs when the server verifies whether the client's UUID is approved. If approved, the server marks the session as active and sends the client's encryption method and necessary credentials. Data Transmission as the client encrypts the sensor data using the specified encryption method and sends it to the server. The server decrypts the data and confirms the successful receipt.

For a returning client, the process is similar but more streamlined. Session ID Validation checks if the client sends the previously received session ID and its UUID. The server checks if the session is valid and the client has reconnected after any ledger updates. Reinitialisation occurs as the client receives the latest ledger information without generating a new session ID if the session ID is still valid. The client can then resume regular operation.

Lastly, if a client with an unapproved UUID attempts to connect to the server, the connection is immediately blocked by stopping communication and marking the session as compromised. No further data is exchanged.

The core functionality of this system revolves around securely transmitting sensor data between the client and server. Before transmission, the data is encrypted using one of several encryption methods (XOR, Caesar, ROT13, SPECK, SIMON), depending on the device's processing power.

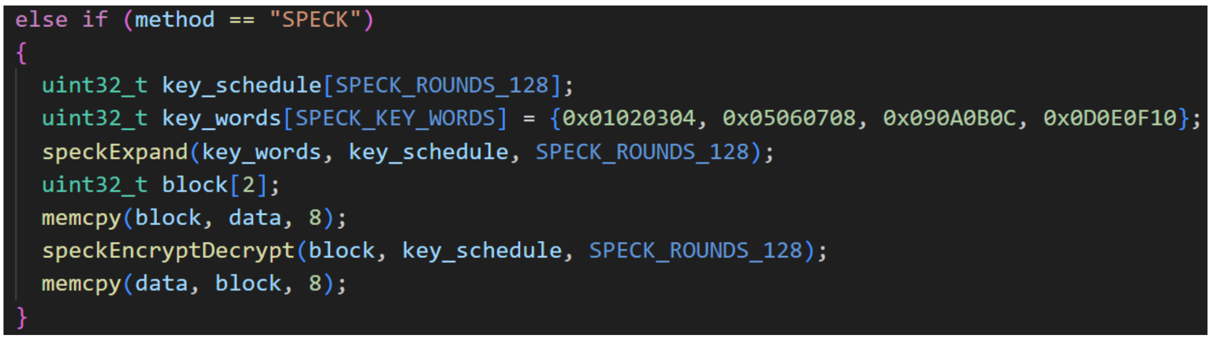
The sensor data preparation occurs as the client gathers sensor data that needs to be sent to the server. In this case, the sensor data is represented as a string; please see *Figure 5.2*.



```c
// Sensor data to be encrypted and sent
char sensorData[] = "The quick brown fox jumps over the lazy dog 1234567890!@#$%^&*()_+-=[]{}|;':,.<>/?\n\t";
```

Figure 5.2: Implementation of Sensor Data Preparation in C

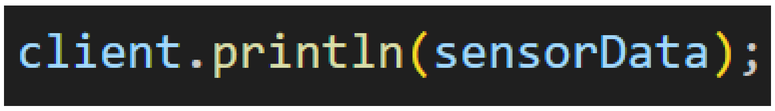
The sensor data is encrypted using a specified encryption method, which could be XOR, Caesar, ROT13, SPECK, or SIMON. Depending on the Adaptive Encryption Methods provided by the Adaptive Amoeba Complexity. The client uses the encryption key provided by the server to perform the encryption. For instance, using the SPECK encryption, the “speckExpand” and “speckEncryptDecrypt” functions handle the key scheduling and encryption processes, as shown in *Figure 5.3*.



```c
else if (method == "SPECK")
{
  uint32_t key_schedule[SPECK_ROUNDS_128];
  uint32_t key_words[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C, 0x0D0E0F10};
  speckExpand(key_words, key_schedule, SPECK_ROUNDS_128);
  uint32_t block[2];
  memcpy(block, data, 8);
  speckEncryptDecrypt(block, key_schedule, SPECK_ROUNDS_128);
  memcpy(data, block, 8);
}
```

Figure 5.3: Implementation of SPECK Encryption Process in C

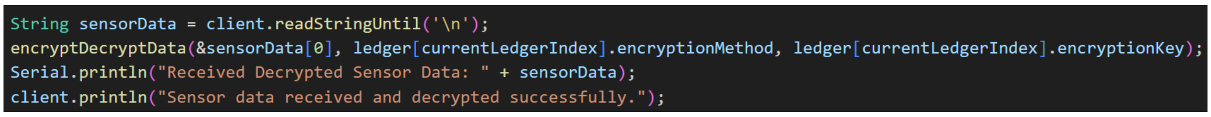
After encryption, the client transmits the encrypted data over the network to the server, as shown in *Figure 5.4*.



```c
client.println(sensorData);
```

Figure 5.4: Implementation of the Data Transmission Process for Secure Sensor Data Exchange in Embedded C.

Upon receiving the data, the server decrypts it using the same method and key as the client, which allows the server to process the decrypted sensor data, as shown in *Figure 5.5*.



```c
String sensorData = client.readStringUntil('\n');
encryptDecryptData(&sensorData[0], ledger[currentLedgerIndex].encryptionMethod, ledger[currentLedgerIndex].encryptionKey);
Serial.println("Received Decrypted Sensor Data: " + sensorData);
client.println("Sensor data received and decrypted successfully.");
```

Figure 5.5: Implementation of the Decryption at the Server Process in C

Mathematically, the encryption process can be represented as:

Equation 5.46: Standard Model of Encryption

$$C = E_K(P)$$

Where:

C is the ciphertext (encrypted data).

P is the plaintext (original sensor data).

E_K is the encryption function with key K.

5.6 Approved Client List Process

When a new device attempts to connect to the server, the client initiates a connection by sending its UUID and processing power details. The server generates a session ID and verifies the UUID against a list of approved UUIDs. If the UUID is approved, the session ID is stored along with other session details. The server then sends the appropriate encryption method, SSID, and password to the client, enabling it to connect securely. The client, in turn, encrypts its sensor data using the provided encryption method and key before sending it to the server.

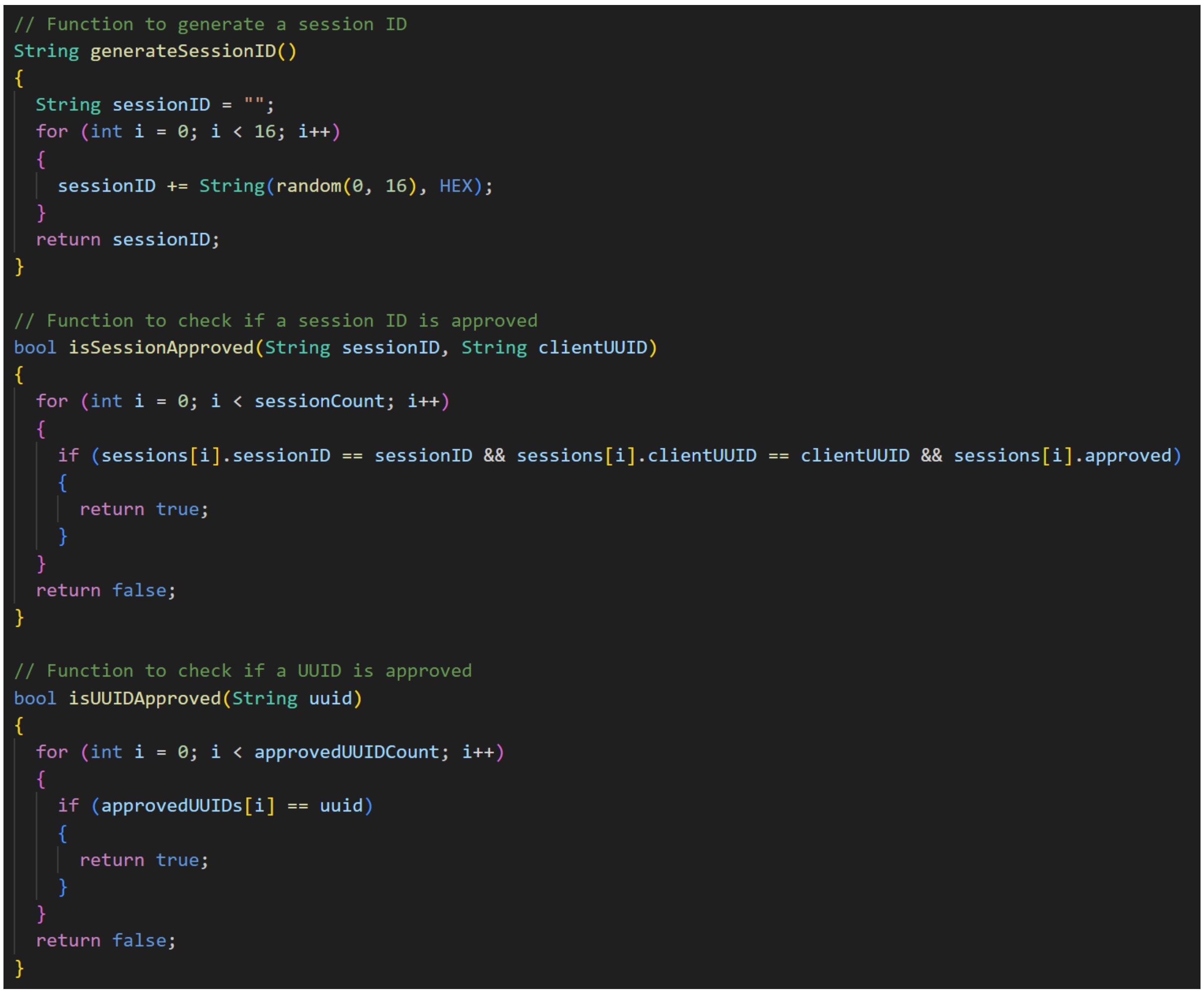
The client uses the stored session ID to re-establish communication with the server for a returning connection. The server checks the validity and expiration of the session IDs. If valid, the client continues its operations without reinitialising the session. The server updates any necessary credentials, and the client resumes its normal operation, including sending encrypted sensor data.

If an unapproved device attempts to connect, the server detects that the UUID is not on the approved list. The server then blocks the session and refuses further communication. The unapproved client cannot connect or send data.

The Approved Client List Process ensures that only authenticated and approved devices can communicate with the server. Each client is identified by a UUID (Universally Unique Identifier), and its approval status is checked during each connection attempt. The use of a session ID provides flexibility and security. The session ID is generated dynamically during each connection and is only valid for a specific session, reducing the risk of replay attacks. Additionally, it allows for session expiration management, where a session can be invalidated after a certain period of inactivity or upon detecting suspicious behaviour. Although MAC addresses are unique identifiers for network devices, they are static and can be spoofed. Using a MAC address alone would make the system vulnerable to spoofing attacks, where an unauthorised device could impersonate an approved device.

The server maintains a list of approved clients identified by UUIDs. Upon a connection request, the server checks whether the client's UUID is in the approved list. If the UUID is approved, the server generates a session ID for that client, which serves as a temporary identifier for managing the session securely. The use of a session ID instead of a MAC address is crucial for enhancing security because MAC addresses are static and can be spoofed, while session IDs are dynamic and unique for each session.

Mathematical Expression: Let $UUID_C$ be the client's UUID, and $UUID_S$ be the set of approved UUIDs. The server checks $UUID_C \in UUID_S$. If true, the server proceeds to generate a session ID, SID, as $SID = GenerateSessionID()$. This session ID is then stored and associated with the client's UUID. The figure below shows the implementation of the UUID and Session IDs. Please see *Figure 5.6*.



```
// Function to generate a session ID
String generateSessionID()
{
  String sessionID = "";
  for (int i = 0; i < 16; i++)
  {
    sessionID += String(random(0, 16), HEX);
  }
  return sessionID;
}

// Function to check if a session ID is approved
bool isSessionApproved(String sessionID, String clientUUID)
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].sessionID == sessionID && sessions[i].clientUUID == clientUUID && sessions[i].approved)
    {
      return true;
    }
  }
  return false;
}

// Function to check if a UUID is approved
bool isUUIDApproved(String uuid)
{
  for (int i = 0; i < approvedUUIDCount; i++)
  {
    if (approvedUUIDs[i] == uuid)
    {
      return true;
    }
  }
  return false;
}
```

Figure 5.6: Implementation of Approved Client List Process Generating and Approving UUID and Session ID functions in C

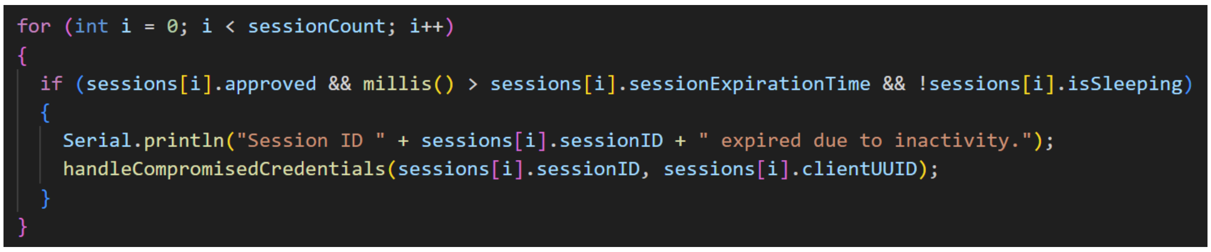
5.7 Client Connection Logic

The client connection logic is designed to manage the connection lifecycle effectively, considering factors like session handling, session expiration, and battery management. The server generates a session ID for each connection, which is used to authenticate and maintain the session. The session expiration time is dynamically adjusted based on the client's battery status. If the battery level is low, the client can enter different sleep modes, and the server adjusts the session expiration time accordingly.

Within session handling, each client session is associated with a unique session ID generated by the server. The server tracks the session's validity and expiration. As implemented in *Figure 5.7*, the server dynamically sets session expiration times based on battery status.

Equation 5.47: Formulating Session Expiration Time

$$sessionExpirationTime = currentTime + sessionDuration$$



```c
for (int i = 0; i < sessionCount; i++)
{
  if (sessions[i].approved && millis() > sessions[i].sessionExpirationTime && !sessions[i].isSleeping)
  {
    Serial.println("Session ID " + sessions[i].sessionID + " expired due to inactivity.");
    handleCompromisedCredentials(sessions[i].sessionID, sessions[i].clientUUID);
  }
}
```

Figure 5.7: Implements the Client Connection Logic from the server to set session expiration time in C

If the session expires or the client goes into sleep mode, the session is marked as inactive. The Battery Management system works as the client's battery management system, continuously monitors the battery voltage, and adjusts the device's operation mode to conserve power:

Equation 5.48: Formulating Battery Percentage

$$batteryPercentage = f(voltage)$$

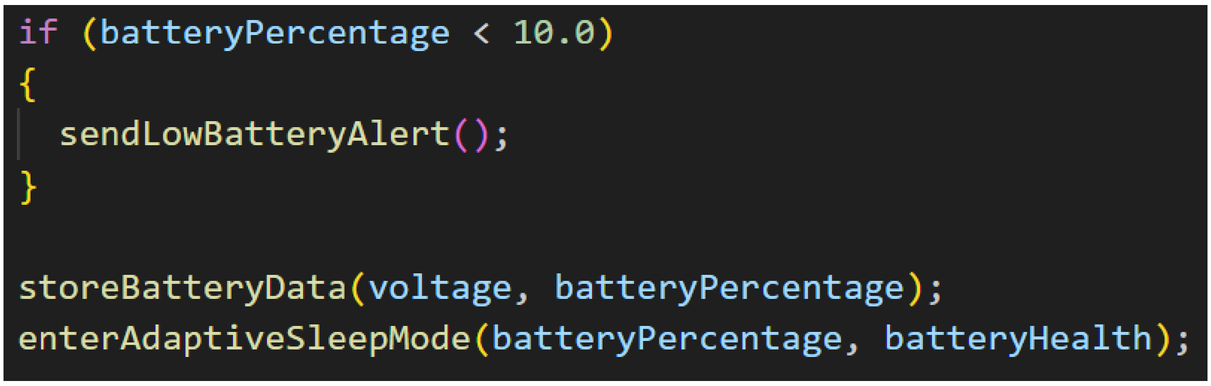
where $f(voltage)$ represents the battery curve mapping function. This function is further explored in the Adaptive Amoeba Battery Curve Mapping Management System.

Depending on the battery percentage, the client enters different sleep modes to conserve energy:

Equation 5.49: Formulating Sleep Duration

$$sleepDuration = g(batteryHealth, batteryPercentage)$$

where $g(\cdot)$ determines the sleep duration based on battery health and remaining capacity. The dynamic session management operates if the client enters any sleep modes, the server is informed, and the session expiration time is adjusted accordingly. The session resumes seamlessly when the client wakes up and reconnects. Please see *Figure 5.8* for the code implementation.



```
if (batteryPercentage < 10.0)
{
  sendLowBatteryAlert();
}

storeBatteryData(voltage, batteryPercentage);
enterAdaptiveSleepMode(batteryPercentage, batteryHealth);
```

Figure 5.8: Implementation of the battery management system adapts the client's behaviour based on the battery percentage function in C

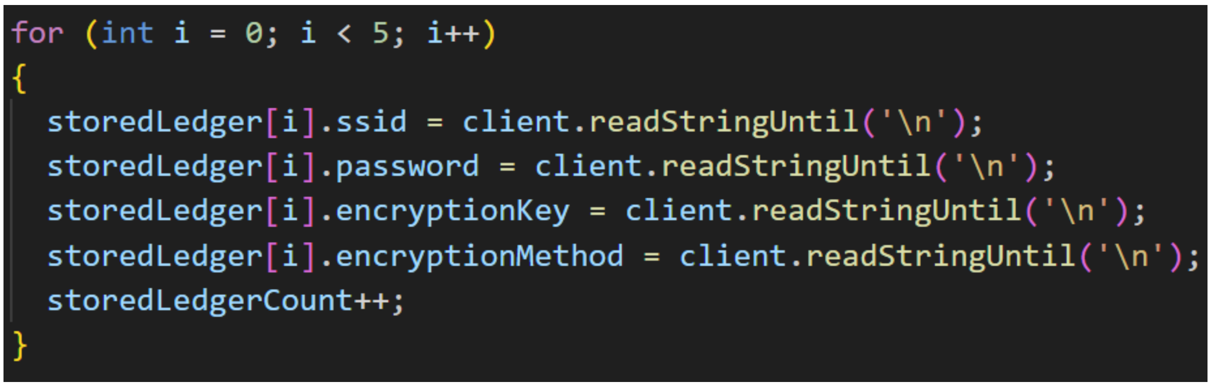
5.8 Shared Ledger Process

The shared ledger process ensures that both the server and client maintain synchronised data regarding the network credentials and encryption methods. The server sends updated ledger information to the client, and the client updates its stored ledger accordingly. The Mathematical Expression: Let L_{server} represent the server's ledger and L_{client} represent the client's ledger. Synchronisation ensures:

Equation 5.50: General Expression for Shared Ledger Process

$$L_{client} \leftarrow L_{server}$$

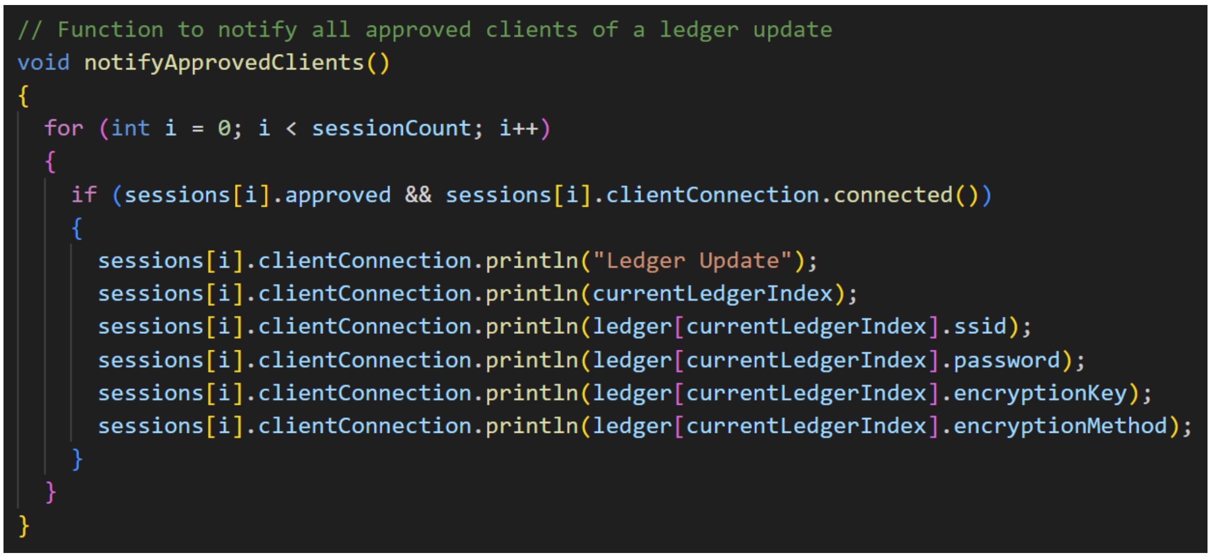
Where L_{client} is updated to match L_{server}. *Figure 5.9* represents the client-slide ledger update.



```
for (int i = 0; i < 5; i++)
{
  storedLedger[i].ssid = client.readStringUntil('\n');
  storedLedger[i].password = client.readStringUntil('\n');
  storedLedger[i].encryptionKey = client.readStringUntil('\n');
  storedLedger[i].encryptionMethod = client.readStringUntil('\n');
  storedLedgerCount++;
}
```

Figure 5.9: Implementation of Client-side ledger update function in C

Figure 5.10 below shows the server-side ledger notification function.



```
// Function to notify all approved clients of a ledger update
void notifyApprovedClients()
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].approved && sessions[i].clientConnection.connected())
    {
      sessions[i].clientConnection.println("Ledger Update");
      sessions[i].clientConnection.println(currentLedgerIndex);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].ssid);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].password);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].encryptionKey);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].encryptionMethod);
    }
  }
}
```

Figure 5.10: Implementation of the server-side ledger notification function in C

5.9 Implementation of Polymorphic Security

Polymorphic security within the Bio-Inspired Lightweight Polymorphic Security System for IoT devices is engineered to provide a dynamic and adaptive approach to securing IoT communication. This system emulates the adaptive and responsive characteristics of biological immune systems, offering a robust mechanism for addressing various security threats, such as unauthorised access, desynchronised states, and compromised clients. By continuously varying encryption methods, session handling strategies and utilising a shared ledger system, this security architecture ensures that the IoT network remains resilient against both known and emerging threats.

The IoT security system closely parallels the core functionalities of the biological immune system. The auto-detection system, much like immune cells detecting pathogens, allows the server to distinguish between approved, unapproved, and desynchronised clients. In a similar vein, the auto-ejection system acts as a defensive measure, rejecting and disconnecting unapproved or compromised clients, mirroring the immune system's role in neutralising threats. Additionally, the trigger ledger system functions analogously to immunological memory, ensuring that all approved clients remain aligned with the same security protocols, thereby maintaining a cohesive and secure network environment.

Both systems emphasise the importance of maintaining a secure and stable environment through dynamic adaptation to new threats, ensuring consistent communication among trusted entities, and promptly addressing any potential security breaches.

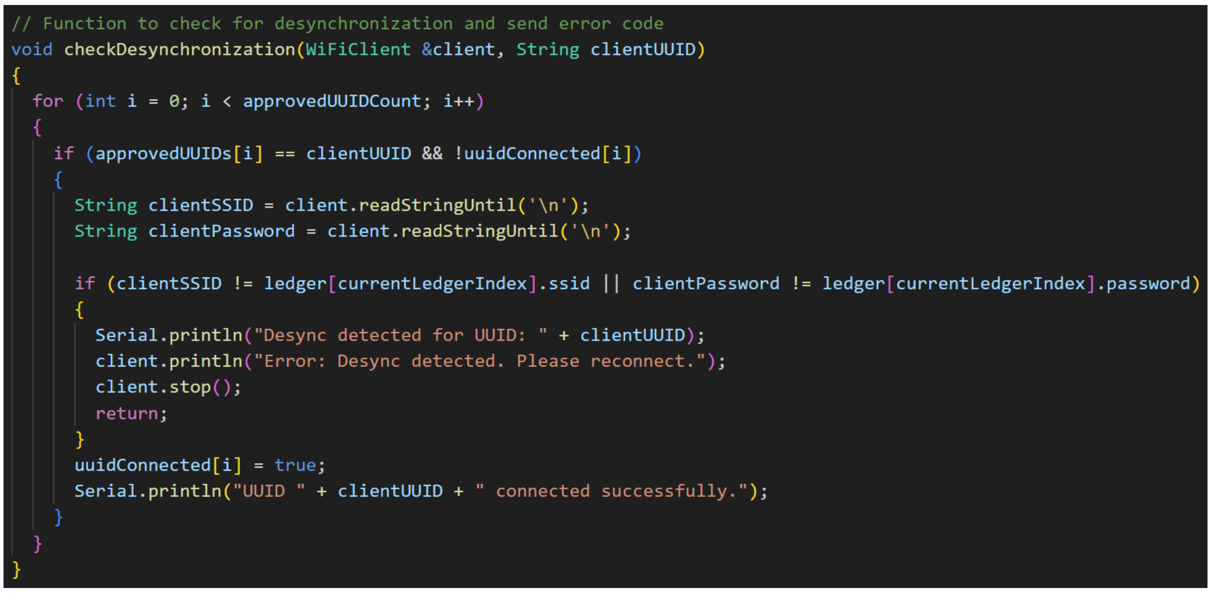
5.9.1 Auto-detection (Immune Surveillance)

The auto-detection system within this security framework functions similarly to the immune system's ability to recognise and differentiate between self and non-self entities. In this context, the server continuously monitors incoming connections and determines whether a client is approved, unapproved, or desynchronised based on predefined criteria.

Approved clients are those with UUIDs that match the entries in the “approvedUUIDs” list. When an approved client attempts to connect, the server verifies the UUID and checks the session ID to determine if the client is in sync with the current state of the shared ledger.

Unapproved clients are immediately identified by the server when their UUID does not match any entry in the “approvedUUIDs” list. These clients are denied access and disconnected to prevent any potential security breach.

Desynchronised clients are those whose stored credentials (SSID and password) do not match the current state of the server's ledger. The server detects this by comparing the incoming client data with the active ledger entry. If a mismatch is found, the server considers the client desynchronised. Please refer to *Figure 5.11* for the desynchronisation function.



```
// Function to check for desynchronization and send error code
void checkDesynchronization(WiFiClient &client, String clientUUID)
{
  for (int i = 0; i < approvedUUIDCount; i++)
  {
    if (approvedUUIDs[i] == clientUUID && !uuidConnected[i])
    {
      String clientSSID = client.readStringUntil('\n');
      String clientPassword = client.readStringUntil('\n');

      if (clientSSID != ledger[currentLedgerIndex].ssid || clientPassword != ledger[currentLedgerIndex].password)
      {
        Serial.println("Desync detected for UUID: " + clientUUID);
        client.println("Error: Desync detected. Please reconnect.");
        client.stop();
        return;
      }
      uuidConnected[i] = true;
      Serial.println("UUID " + clientUUID + " connected successfully.");
    }
  }
}
```

Figure 5.11: Implementation to check for any approved client desynchronisation function in C

In this system, the server reads the incoming client data and compares it with the expected data in the ledger. If the client data does not match, a desynchronisation error is triggered, and the client is instructed to reconnect.

The mathematical representation of the approved client list is as follows. Let $UUID_c$ be the UUID of the connecting client, and let $UUID_s$ represent the set of approved UUIDs in the system. The detection logic can be represented as:

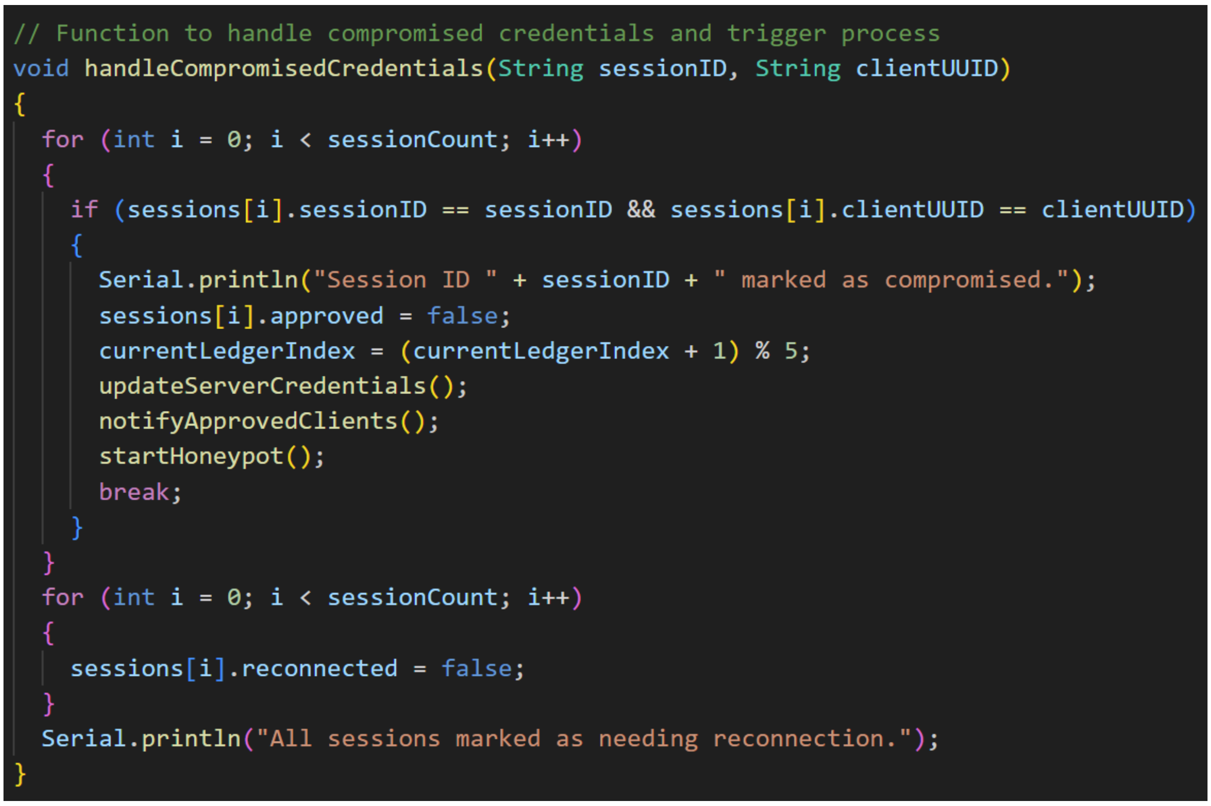
Equation 5.51: Client Status Detection Logic

$$Status(c) = \left\{ \begin{matrix} Approved & if\ UUID_c \in UUID_s \\ Unapproved & if\ UUID_c \notin UUID_s \\ Desynced & if\,(SSID_c, Password_c) \neq (SSID_s, Password_s) \end{matrix} \right\}$$

Where $(SSID_c, Password_c)$ is the client's submitted credentials and $(SSID_s, Password_s)$ is the current valid entry in the server's ledger.

5.9.2 Auto-ejection (Immune Response)

The auto-ejection system is akin to the immune system's mechanism of eliminating foreign or harmful entities. This system ensures that any unapproved or compromised clients are automatically disconnected from the network, maintaining the integrity and security of the IoT environment. Approved clients remain connected as long as they comply with the session management rules, such as maintaining synchronisation with the server's ledger and responding within the session expiration time. These clients are automatically ejected as soon as they are detected. Once their unapproved status is confirmed, the server does not allow them to communicate further. Desynchronised clients are either instructed to reconnect (if approved but desynchronised) or are ejected from the network if their desynchronisation cannot be resolved. Please see *Figure 5.12*.



```
// Function to handle compromised credentials and trigger process
void handleCompromisedCredentials(String sessionID, String clientUUID)
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].sessionID == sessionID && sessions[i].clientUUID == clientUUID)
    {
      Serial.println("Session ID " + sessionID + " marked as compromised.");
      sessions[i].approved = false;
      currentLedgerIndex = (currentLedgerIndex + 1) % 5;
      updateServerCredentials();
      notifyApprovedClients();
      startHoneypot();
      break;
    }
  }
  for (int i = 0; i < sessionCount; i++)
  {
    sessions[i].reconnected = false;
  }
  Serial.println("All sessions marked as needing reconnection.");
}
```

Figure 5.12: Implementation of the auto ejection system through the handle compromised credentials function in C

In this implementation, when a session is marked as compromised, the client is effectively ejected by invalidating the session and updating the server's credentials. The system also notifies all approved clients of the update and may activate the honeypot to handle any desynchronised clients.

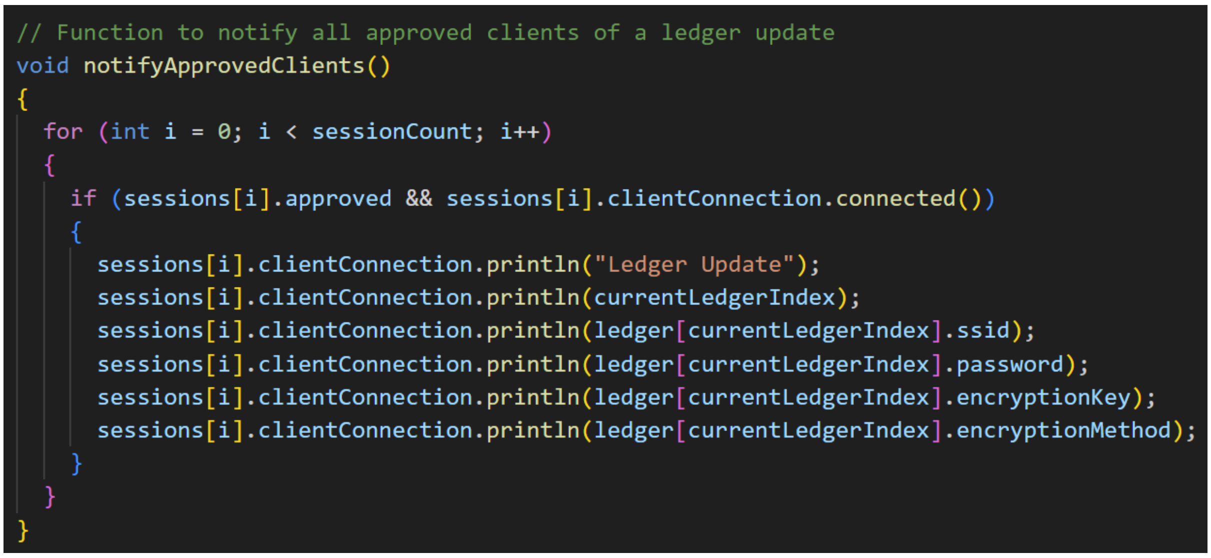
The mathematical representation of the auto ejection system is as follows. Let $S(t)$ represent the session state at time t. The ejection logic can be expressed as:

Equation 5.52: Auto Ejection Logic System

$$S(t+1) = \begin{cases} Ejected & if\ UUID_c \notin UUID_s\ or\ compromised \\ Reconnected & if\ UUID_c \in UUID_s\ and\ resynchronised \\ S(t) & if\ session\ remains\ valid \end{cases}$$

5.9.3 Trigger Ledger (Memory Response)

The shared ledger serves as a collective memory within the network, ensuring all approved clients operate under consistent security credentials. This system is comparable to the immune system's memory cells, which retain information about previously encountered pathogens for faster response upon re-exposure. The ledger acts as a source of truth for approved clients, providing up-to-date credentials and encryption keys that ensure secure communication. Upon any update to the ledger, all approved clients are notified, and the ledger is synchronised across the network. Unapproved clients cannot access the ledger and, therefore, cannot synchronise or communicate securely with the network. Desynchronised clients rely on the ledger to resynchronise their credentials. If a client is desynchronised, the server may provide updated credentials or prompt the client to reconnect with the correct information.



```c
// Function to notify all approved clients of a ledger update
void notifyApprovedClients()
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].approved && sessions[i].clientConnection.connected())
    {
      sessions[i].clientConnection.println("Ledger Update");
      sessions[i].clientConnection.println(currentLedgerIndex);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].ssid);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].password);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].encryptionKey);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].encryptionMethod);
    }
  }
}
```

Figure 5.13: Implementation of the trigger for the ledger function in C

The server pushes ledger updates to all connected and approved clients, ensuring they have the latest credentials and encryption methods. Please see *Figure 5.13*. The ledger synchronisation can be mathematically expressed as:

Equation 5.53: Ledger Synchronisation

$$L(t+1) = L(t) + \Delta L$$

Where $L(t)$ represents the ledger state at the time t and ΔL represents the change (update) to the ledger. All approved clients receive this update, represented as:

Equation 5.54: Updating All Approved Clients

$$Ci(t+1) = Ci(t) \cup \Delta L$$

Where $Ci(t)$ represents the credential state for the client i at time t, and ΔL is the ledger update.

5.10 Adaptive Amoeba Battery Curve Mapping Management System

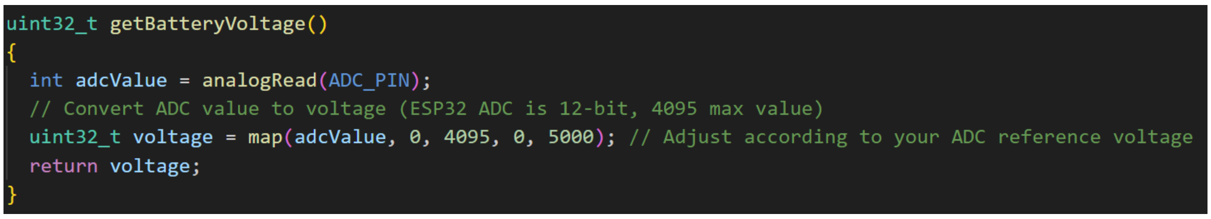
In Figure 4.2, the block diagram for the Adaptive Amoeba Battery Curve Mapping Management System encapsulates the detailed workings of the battery management system, from initial voltage measurement through adaptive sleep mode selection and eventual hibernation. This ensures efficient power usage and extended operational life for IoT devices. First, it is crucial to break down the key processes to program this system for implementation.

Firstly, there needs to be a conversion of an analogue-to-digital converter (ADC) reading to a corresponding voltage value:

Equation 5.55: Formulation for Voltage Battery Mapping

$$V_{battery} = map(ADC\ value, 0{,}4095{,}0{,}5000)$$

The *ADC value* is the raw reading from the ADC pin (0 to 4095 for a 12-bit ADC). $V_{battery}$ is the battery voltage in millivolts. When programming the formula in C, please see *Figure 5.14*.



```c
uint32_t getBatteryVoltage()
{
  int adcValue = analogRead(ADC_PIN);
  // Convert ADC value to voltage (ESP32 ADC is 12-bit, 4095 max value)
  uint32_t voltage = map(adcValue, 0, 4095, 0, 5000); // Adjust according to your ADC reference voltage
  return voltage;
}
```

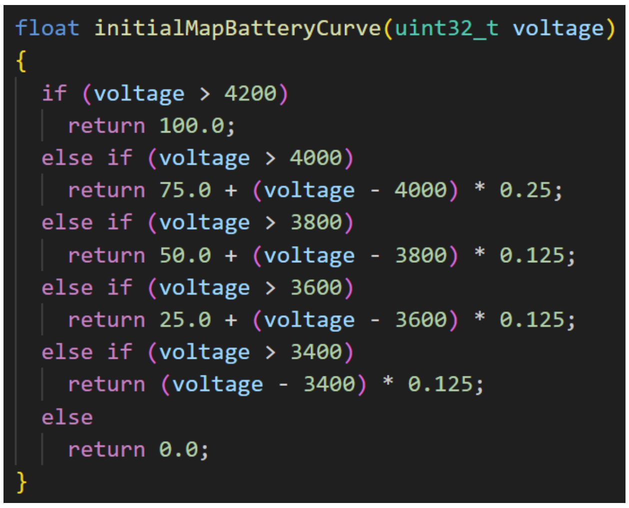
Figure 5.14: Implementation of the Battery Voltage function in C

Secondly, the initial battery percentage mapping can provide an initial estimate of the battery percentage based on the voltage, which can be represented as a piecewise function:

Equation 5.56: Estimation of Battery Percentage

$$P_{initial} = \left\{ \begin{array}{ll} 100.0 & if\ V_{battery} > 4200 \\ 75.0 + 0.25 \times (V_{battery} - 4000) & if\ 4000 < V_{battery} \leq 4200 \\ 50.0 + 0.125 \times (V_{battery} - 3800) & if\ 4000 < V_{battery} \leq 4000 \\ 25.0 + 0.125 \times (V_{battery} - 3600) & if\ 4000 < V_{battery} \leq 3800 \\ 0.125 \times (V_{battery} - 3400) & if\ 4000 < V_{battery} \leq 3600 \\ 0.0 & if\ V_{battery} \leq 3400 \end{array} \right\}$$

When programming this piecewise function in C, please see *Figure 5.15*.



```c
float initialMapBatteryCurve(uint32_t voltage)
{
  if (voltage > 4200)
    return 100.0;
  else if (voltage > 4000)
    return 75.0 + (voltage - 4000) * 0.25;
  else if (voltage > 3800)
    return 50.0 + (voltage - 3800) * 0.125;
  else if (voltage > 3600)
    return 25.0 + (voltage - 3600) * 0.125;
  else if (voltage > 3400)
    return (voltage - 3400) * 0.125;
  else
    return 0.0;
}
```

Figure 5.15: Implementation of the initial battery percentage mapping piecewise function in C

The next feature is to store the battery data using a circular buffer with a dynamic EEPROM management approach to store the battery voltage and corresponding percentage:

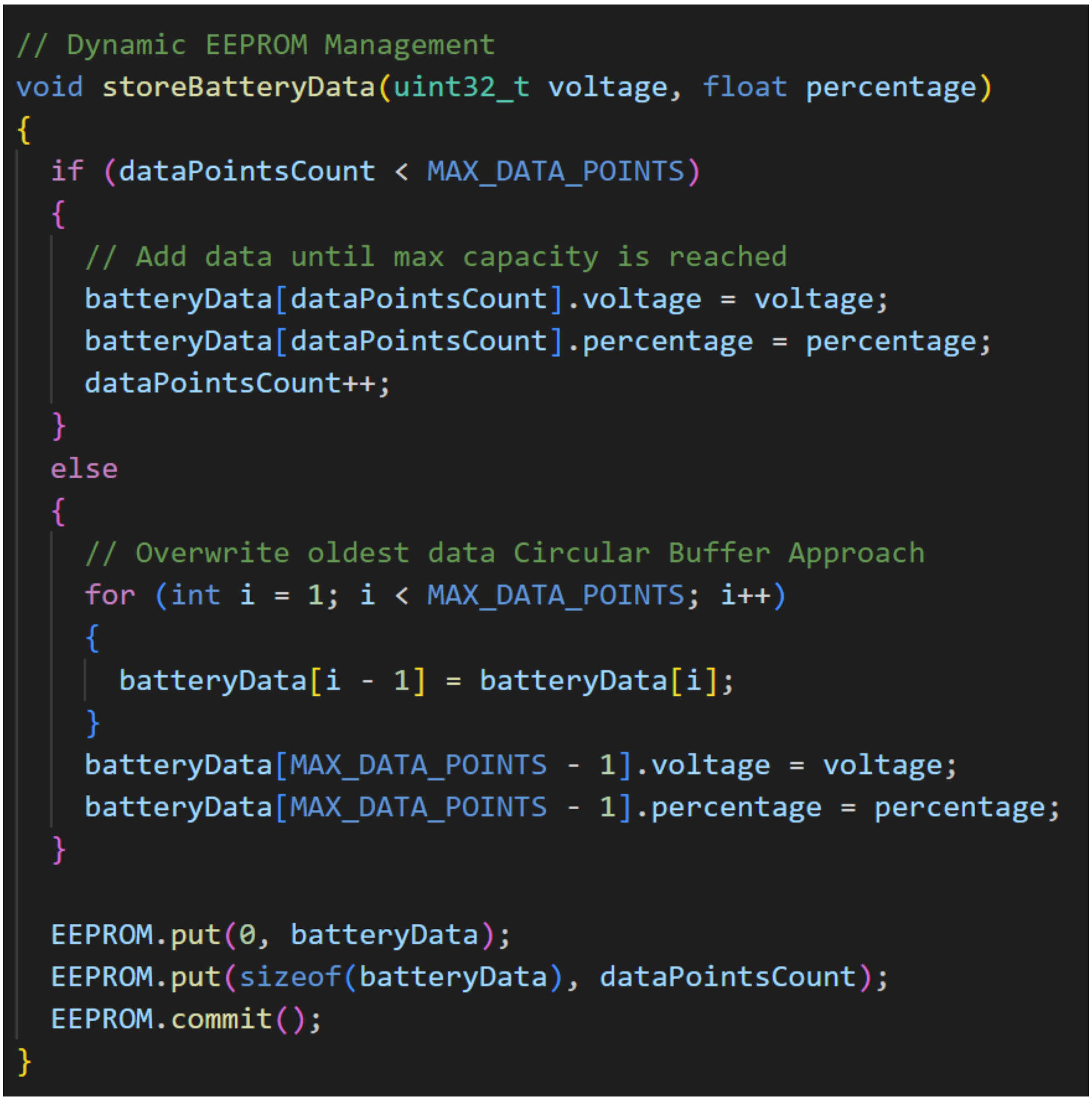
Equation 5.57: Circular Buffer With Dynamic EEPROM Management

$$batteryData[i].voltage = V_{battery}$$

$$batteryData[i].precentage = P_{current}$$

$$batteryData[i-1] = batteryData[i]$$

The oldest data will be overwritten if the buffer reaches its maximum capacity. Please see *Figure 5.16* for the function represented in C.



```c
// Dynamic EEPROM Management
void storeBatteryData(uint32_t voltage, float percentage)
{
  if (dataPointsCount < MAX_DATA_POINTS)
  {
    // Add data until max capacity is reached
    batteryData[dataPointsCount].voltage = voltage;
    batteryData[dataPointsCount].percentage = percentage;
    dataPointsCount++;
  }
  else
  {
    // Overwrite oldest data Circular Buffer Approach
    for (int i = 1; i < MAX_DATA_POINTS; i++)
    {
      batteryData[i - 1] = batteryData[i];
    }
    batteryData[MAX_DATA_POINTS - 1].voltage = voltage;
    batteryData[MAX_DATA_POINTS - 1].percentage = percentage;
  }

  EEPROM.put(0, batteryData);
  EEPROM.put(sizeof(batteryData), dataPointsCount);
  EEPROM.commit();
}
```

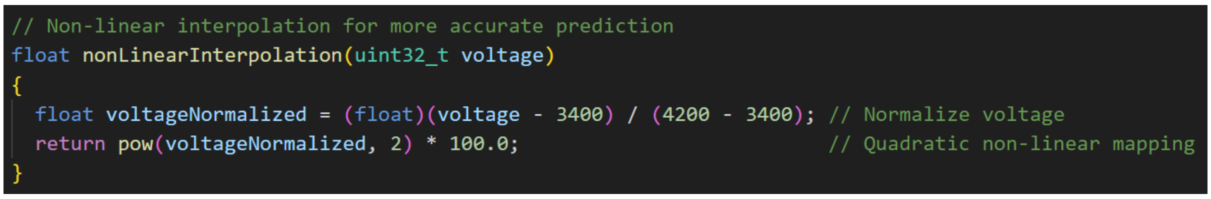
Figure 5.16: Implementation of the Storing Battery Data function in C

Once the battery data is stored, it can be loaded from the EEPROM anytime the data needs to be used, especially in the next feature, which will be used to predict the battery percentage. The system uses non-linear interpolation for a more accurate battery percentage prediction:

Equation 5.58: Non-Linear Interpolation for Battery Percentage Prediction

$$P_{predicted} = \left(\frac{V_{battery} - 3400}{4200 - 3400}\right)^2 \times 100$$

Where $V_{battery}$ is normalised between 3400mV and 4200mV. The normalised value is squared to apply a quadratic mapping, resulting in a non-linear curve that better reflects typical battery discharge behaviour. Please see *Figure 5.17* for the function represented in C.



```c
// Non-linear interpolation for more accurate prediction
float nonLinearInterpolation(uint32_t voltage)
{
  float voltageNormalized = (float)(voltage - 3400) / (4200 - 3400); // Normalize voltage
  return pow(voltageNormalized, 2) * 100.0;                          // Quadratic non-linear mapping
}
```

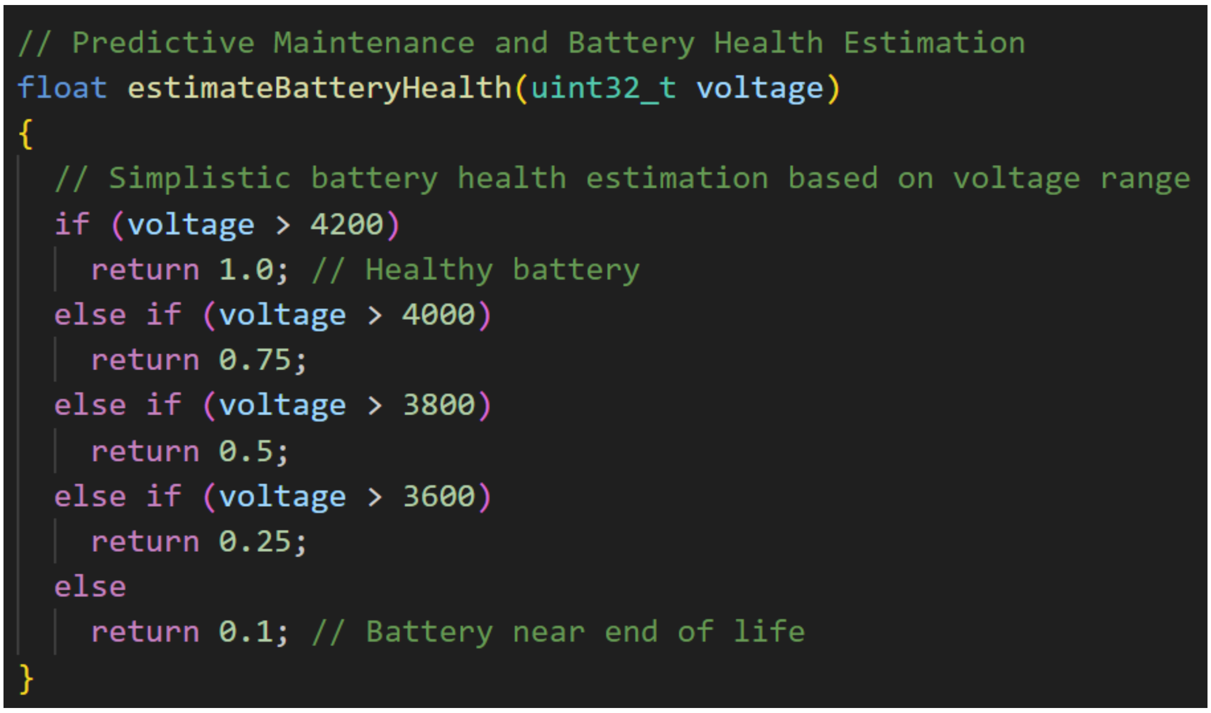
Figure 5.17: Implementation of the Predicting Battery Percentage using a non-linear interpolation function in C

Additionally, this system looks at the battery health based on the voltage once the system is fully charged or what it reads as its maximum voltage:

Equation 5.59: Formulation for Battery Health

$$H_{battery} = \left\{ \begin{array}{ll} 1.0 & if\ V_{battery} > 4200 \\ 0.75 & if\ 4000 < V_{battery} \leq 4200 \\ 0.5 & if\ 3800 < V_{battery} \leq 4000 \\ 0.25 & if\ 3600 < V_{battery} \leq 3800 \\ 0.10 & if\ V_{battery} \leq 3600 \end{array} \right\}$$

Where $H_{battery}$ is the battery health factor, ranging from 0.1 (poor health) to 1.0 (excellent health). Please see *Figure 5.18* for the function represented in C.



```c
// Predictive Maintenance and Battery Health Estimation
float estimateBatteryHealth(uint32_t voltage)
{
  // Simplistic battery health estimation based on voltage range
  if (voltage > 4200)
    return 1.0; // Healthy battery
  else if (voltage > 4000)
    return 0.75;
  else if (voltage > 3800)
    return 0.5;
  else if (voltage > 3600)
    return 0.25;
  else
    return 0.1; // Battery near end of life
}
```

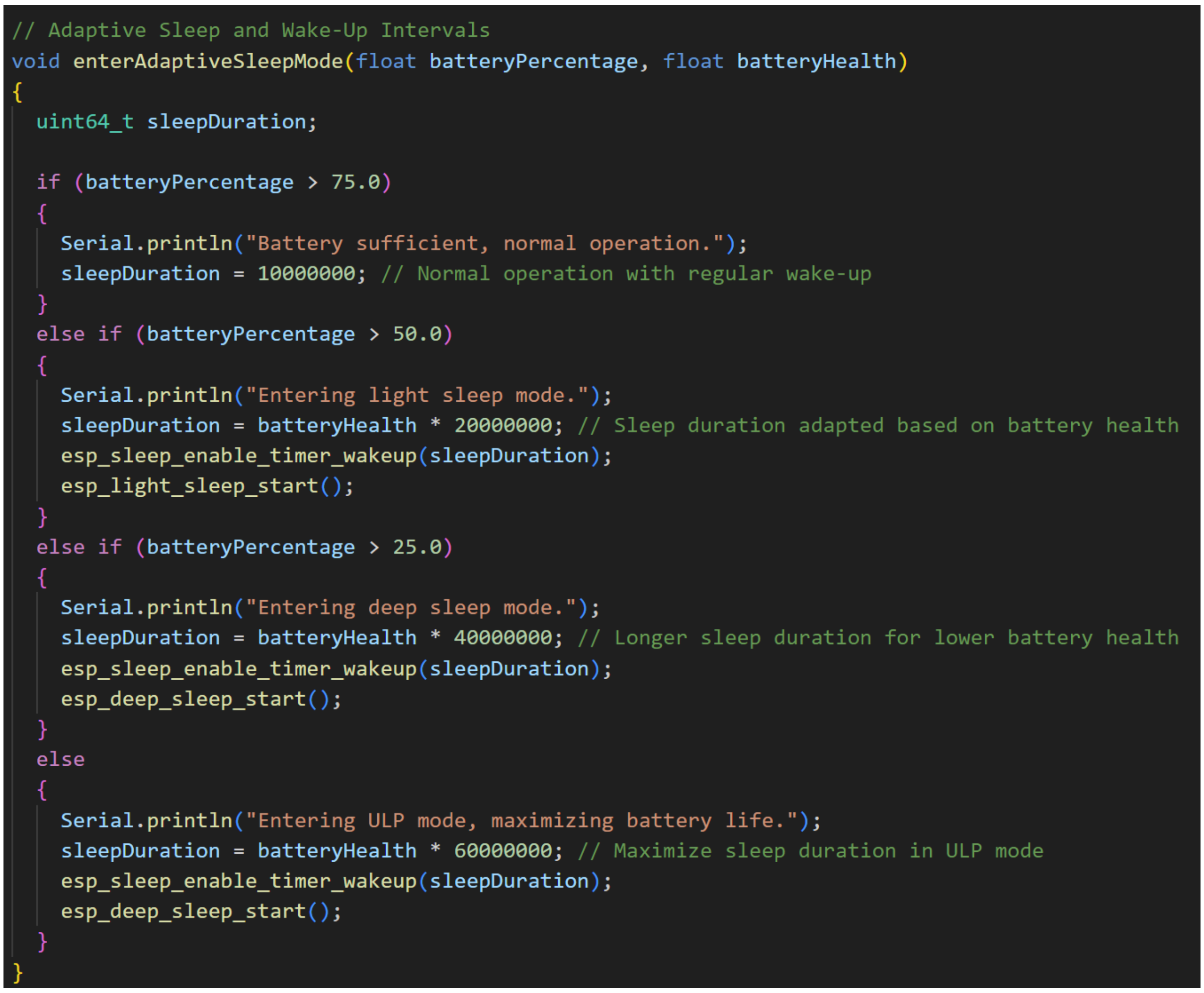
Figure 5.18: Implementation of the predictive battery health function in C

Lastly, the implementation of the adaptive sleep mode duration based on the battery percentage and health, which can be easily configurable depending on the desired threshold levels:

Equation 5.60: Formulation for Adaptive Sleep Modes

$$Sleep\ Mode = \left\{ \begin{array}{ll} 10^7 & if\ P_{predicted} > 75.0\ (normal\ operation) \\ 2 \times 10^7 \times H_{battery} & if\ 50.0 < \ P_{predicted} \leq 75.0\ (lighth\ sleep) \\ 4 \times 10^7 \times H_{battery} & if\ 25.0 < P_{predicted} \leq 50.0\ (deep\ sleep) \\ 6 \times 10^7 \times H_{battery} & if\ P_{predicted} \leq 25.0\ (ULP) \end{array} \right\}$$

Which can be represented by the following function in *Figure 5.19*.



```c
// Adaptive Sleep and Wake-Up Intervals
void enterAdaptiveSleepMode(float batteryPercentage, float batteryHealth)
{
  uint64_t sleepDuration;

  if (batteryPercentage > 75.0)
  {
    Serial.println("Battery sufficient, normal operation.");
    sleepDuration = 10000000; // Normal operation with regular wake-up
  }
  else if (batteryPercentage > 50.0)
  {
    Serial.println("Entering light sleep mode.");
    sleepDuration = batteryHealth * 20000000; // Sleep duration adapted based on battery health
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_light_sleep_start();
  }
  else if (batteryPercentage > 25.0)
  {
    Serial.println("Entering deep sleep mode.");
    sleepDuration = batteryHealth * 40000000; // Longer sleep duration for lower battery health
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_deep_sleep_start();
  }
  else
  {
    Serial.println("Entering ULP mode, maximizing battery life.");
    sleepDuration = batteryHealth * 60000000; // Maximize sleep duration in ULP mode
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_deep_sleep_start();
  }
}
```

Figure 5.19: Implementation of the sleep mode function in C

Do note that the difference between deep sleep and ultra-low power mode is that the time in microseconds can also be adapted to wake up only after certain other conditions are met, such as waking on LAN for connected devices or only waking up when the sensor data is significantly different. With every loop cycle, the system performs the following:

- Measure the current battery voltage $V_{battery} = map(ADC\ value)$.
- Predict the current battery percentage $P_{predicted}$ Using either the initial map or non-linear interpolation from stored data.
- Estimates the battery health $H_{battery}$.

- Store the voltage and predicted percentage in EEPROM.
- Dynamically adjusts the device's operation mode based on $P_{predicted}$ and $H_{battery}$ determining T_{sleep}.
- Enter the appropriate sleep mode if needed.
- Loops back after the delay or other normal operations for the next reading and adjustment.

Overall, the code systematically adjusts the power management of an IoT device by mapping the ADC voltage reading to a battery percentage, refining the prediction through non-linear interpolation, and dynamically managing sleep durations based on battery health:

Equation 5.61: General Formulations for Power Management

$$Voltage\ to\ Precentate\ Mapping: P_{predicted} = f\left(V_{battery}\right)$$

$$Battery\ Health\ Estimation: H_{battery} = g\left(V_{battery}\right)$$

$$Adaptive\ Sleep\ Duration: T_{sleep} = h(P_{predicted}, H_{battery})$$

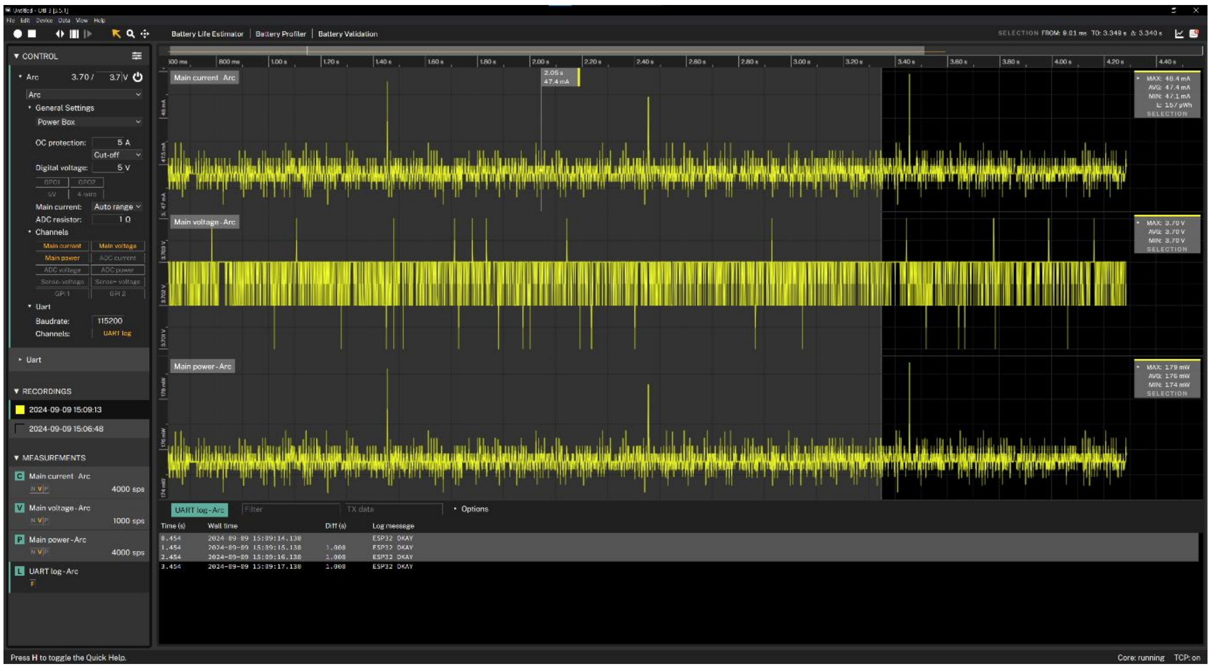
Where f represents the initial mapping and non-linear interpolation functions. g represents the battery health estimation function. h represents the function determining adaptive sleep duration based on $P_{predicted}$ and $H_{battery}$. This process ensures that the device adapts its power consumption based on real-time battery data, continuously refining its understanding of the battery's discharge curve through data collection and interpolation. Please see *Appendix 14* for the complete documentation of the Adaptive Amoeba Battery Curve Mapping Management System.

The encryption switching logic embedded within the AABCMS serves not only as a power-saving technique but also as a security modulation system. As lighter algorithms are more susceptible to attack, their use is reserved strictly for low-energy conditions where computational economy is prioritised over resilience. Conversely, in high-energy states, algorithms such as SPECK and SIMON offer superior cryptographic robustness, mirroring the strategic deployment of full immune responses in organisms under stress. This hierarchical cryptographic adaptation embodies a form of bio-inspired polymorphic security behaviour.

6.0 Chapter Six: Testing and Refinement

The preceding chapter demonstrated the full implementation of the bio-inspired lightweight polymorphic security system for IoT devices, both through C programming and the corresponding mathematical models. This chapter is dedicated to the comprehensive testing and refinement of the system, focusing on elucidating the reasoning behind key design decisions. It presents a systematic evaluation of the system's performance across diverse operational scenarios, offering an in-depth analysis of its framework and responsiveness to varying conditions. Special attention is given to assessing the efficiency and functionality of critical subsystems, with a particular focus on the Adaptive Amoeba Battery Curve Mapping Management System. Power analysis tools are employed to rigorously examine the system's energy consumption and optimisation, underscoring its effectiveness and resilience in environments constrained by power. The primary testing will be conducted using the ESP32-S3 development board, a 32-bit microcontroller (detailed in Appendix 3), with comparative analysis against a lower-grade 8-bit ATMEGA328P microcontroller, demonstrating the system's adaptability, efficiency, and overall robustness across differing hardware configurations.

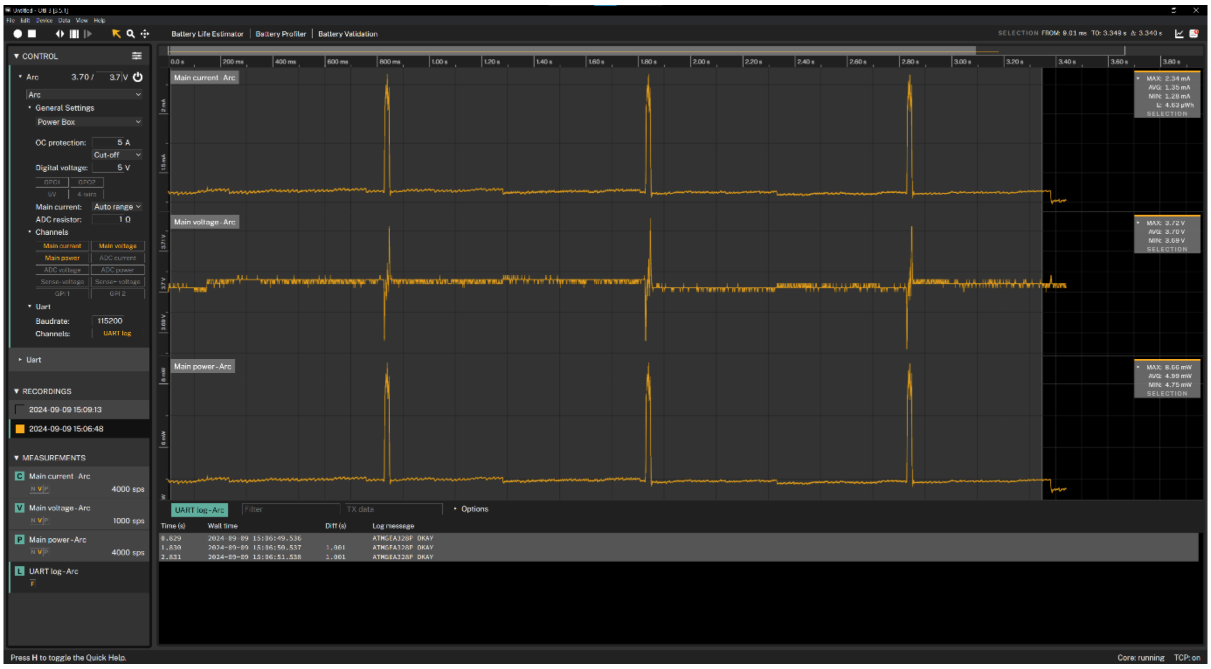
The control should be established before going into the different subchapters to test different aspects of the framework. The control is how the microcontroller operates with a simple algorithm that uses the serial to print a statement every second. Baseline power consumption for each device is demonstrated in *Figure 6.1*, which shows that the ESP32-S3 has an average current draw of 47.4mA at 3.7V with 176mW.



```
Time (s)   Wall time                  Diff (s)   Log message
0.454      2024-09-09 15:39:14.138               ESP32 OKAY
1.454      2024-09-09 15:39:15.138    1.000      ESP32 OKAY
2.454      2024-09-09 15:39:16.138    1.000      ESP32 OKAY
3.454      2024-09-09 15:39:17.138    1.000      ESP32 OKAY
```

Figure 6.1: Otii Power Analysis Demonstrating Baseline Power Consumption for the ESP32-S3 Running the Control Algorithm.

Figure 6.2 shows that the ATMEGA328P has an average current draw of 1.35mA at 3.7V with 4.99mW.



```
Time (s)   Wall time                  Diff (s)   Log message
0.829      2024-09-09 15:36:49.536               ATMGEA328P OKAY
1.830      2024-09-09 15:36:50.537    1.001      ATMGEA328P OKAY
2.831      2024-09-09 15:36:51.538    1.001      ATMGEA328P OKAY
```

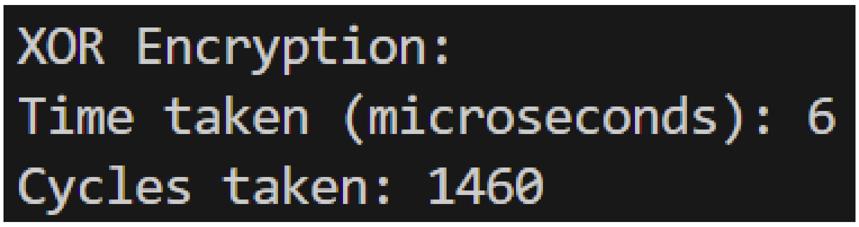
Figure 6.2: Otii Power Analysis Demonstrating Baseline Power Consumption for the ATMEGA328P

6.1 Testing the Efficiency of Encryption Methods Implemented

The following subchapter will test the different encryption methods used in this framework and measure the number of cycles and power it takes to complete both encryption and decryption algorithms. The testing will be done for ESP32-S3 (running at 240MHz) and ATMEGA328P (running at 16MHz) processors to see and have a comparative difference between 8-bit and 32-bit processors. Please refer to Appendix 13, Cycle Testing for Encryption Methods.

6.1.1 Cycle Testing XOR Encryption

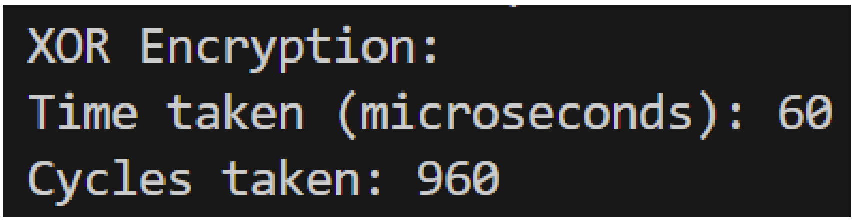
ESP32-S3 XOR cycle encryption testing shows that the time taken in microseconds is 6µS and 1460 cycles taken to complete this; please see *Figure 6.3*.



```
XOR Encryption:
Time taken (microseconds): 6
Cycles taken: 1460
```

Figure 6.3: ESP32-S3 XOR cycle testing serial print output

In comparison, the ATMEGA328P cycle encryption testing shows that the time taken in microseconds is 60µS and 960 cycles are taken to complete this; please see *Figure 6.4*.



```
XOR Encryption:
Time taken (microseconds): 60
Cycles taken: 960
```

Figure 6.4: ATMEGA328P XOR cycle testing serial print output

The difference in cycle counts for the XOR encryption between the ESP32 and ATMEGA328P microcontrollers, despite the time difference, can be attributed to the underlying architecture of these processors, the clock speed, and how they handle instructions.

The ESP32 typically operates at a much higher clock speed (up to 240 MHz). The ATMEGA328P, on the other hand, has a lower clock speed (typically 16 MHz). Higher clock speed means the ESP32 completes each instruction faster (in terms of time), but it might also involve additional cycles due to how the architecture handles specific tasks, especially in microcontroller designs with more complex instruction sets.

The ESP32 uses the Tensilica Xtensa architecture, which is more complex than the AVR architecture used in the ATMEGA328P. As a result, even though the ESP32 has a higher clock speed and may take less time for individual operations, the cycles needed for specific instructions, such as XOR, could be more due to the complexity of the processing or how instructions are decoded and executed. In contrast, the ATMEGA328P has a simpler architecture (AVR 8-bit RISC), where instructions like XOR are likely implemented more directly in fewer cycles, even though the overall processing speed is slower.

The ESP32 is a dual-core processor and has a deeper pipeline than the ATMEGA328P. This means that while it can run more operations simultaneously, certain simple operations like XOR encryption might involve more overhead due to how the pipeline or multicore system is managed, leading to more cycles. The ATMEGA328P's simpler design results in fewer cycles, even if it’s slower in absolute time, as it has less overhead.

The cycles are calculated based on the number of clock ticks required to execute the instruction. Mathematically, for a simple XOR instruction:

Equation 6.1: Calculation for Clock Ticks required to Execute Instruction for XOR

$$Total\ Time\ (in\ seconds) = \frac{Cycle\ Count}{Clock\ Speed\ (Hz)}$$

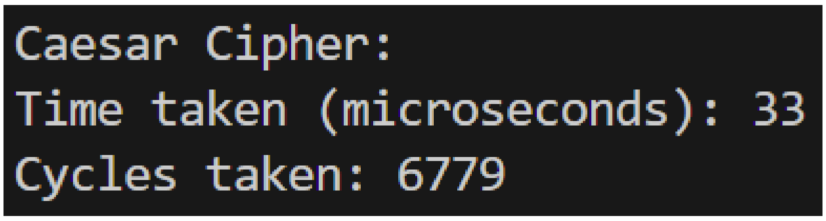
$$Time\ for\ ESP32\ (microseconds) = \frac{1460}{240{,}000{,}000} \times 1{,}000{,}000 = 6\mu S$$

$$Time\ for\ ATMEGA328P\ (microseconds) = \frac{960}{16{,}000{,}000} \times 1{,}000{,}000 = 60\mu S$$

Thus, even though the ESP32 has more cycles (1460) for the XOR encryption, its higher clock speed compensates for this, making the total time much lower (6 µs) compared to the ATMEGA328P (60 µs).

6.1.2 Cycle Testing Caesar Cipher

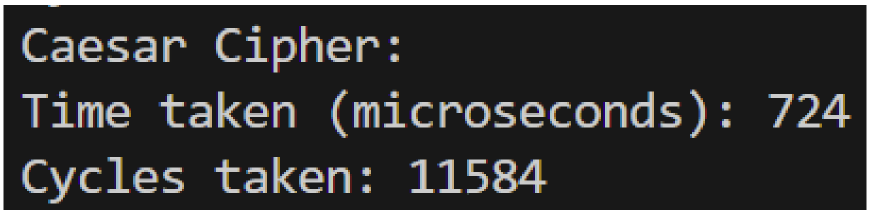
ESP32-S3 Caesar cycle encryption testing shows that the time taken in microseconds is 33µS, and 6779 cycles are required to complete this; please see *Figure 6.5*.



```
Caesar Cipher:
Time taken (microseconds): 33
Cycles taken: 6779
```

Figure 6.5: ESP32-S3 Caesar cycle testing serial print output

In comparison, the ATMEGA328P cycle encryption testing shows that the time taken in microseconds is 724µS and 115284 cycles to complete this; please see *Figure 6.6*.



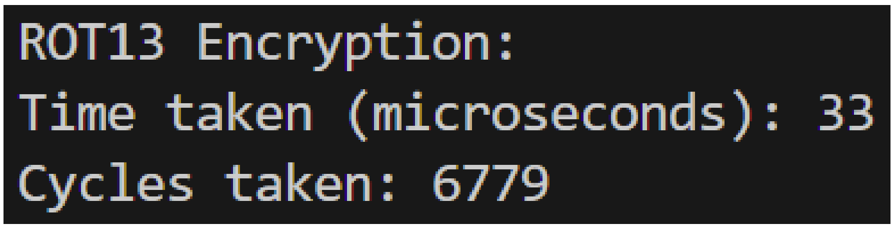
```
Caesar Cipher:
Time taken (microseconds): 724
Cycles taken: 11584
```

Figure 6.6: ATMEGA328P Casear cycle testing serial print output

As the encryption methods are getting more complex, there is a clear disparity between the 8-bit and 32-bit processors.

6.1.3 Cycle Testing ROT13 Encryption

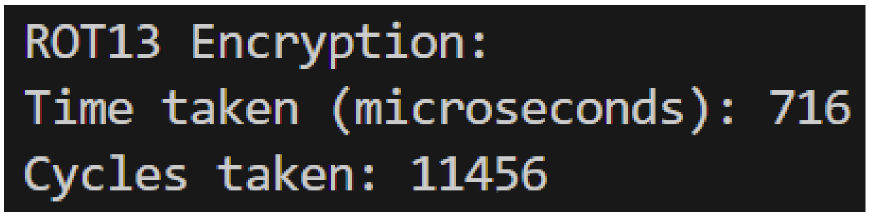
ESP32-S3 Caesar cycle encryption testing shows that the time taken in microseconds is 33µS, and 6779 cycles are required to complete this; please see *Figure 6.7*.



```
ROT13 Encryption:
Time taken (microseconds): 33
Cycles taken: 6779
```

Figure 6.7: ESP32-S3 ROT13 cycle testing serial print output

In comparison, the ATMEGA328P ROT13 cycle encryption testing shows that the time taken in microseconds is 716µS and 11456 cycles to complete this; please see *Figure 6.8*.



```
ROT13 Encryption:
Time taken (microseconds): 716
Cycles taken: 11456
```

Figure 6.8: ATMEGA328P ROT13 cycle testing serial print output

The reason for both the Caesar cipher and ROT13 encryption methods taking the same time and cycles on the ESP32 is that these two encryption algorithms are very similar in terms of their fundamental operations. Both Caesar cipher and ROT13 are shift ciphers, where the primary operation is shifting each letter in the plaintext by a specific number of positions in the alphabet. ROT13 is a specific case of the Caesar cipher where the shift is always 13 positions, meaning that every letter is shifted by exactly half of the alphabet's length. Caesar cipher uses a variable shift (which can be any number of positions), but when the shift is set to 13, it behaves exactly like ROT13. In the code, both encryption methods involve shifting the characters of the input string based on the encryption key. For ROT13, the key is fixed at 13, while for Caesar, the key can be any number.

Both algorithms perform a character-by-character shift within the alphabet, and this shift operation is computationally very simple, involving only an arithmetic operation (addition or subtraction). In terms of actual computation, shifting by 13 positions in ROT13 and shifting by a user-defined number in Caesar cipher will take an identical number of operations, especially since the range of shifts in the alphabet (26 letters) is small and uniform.

Regarding code structure, the Caesar and ROT13 functions you are using likely share much of the same logic, with the only difference being the input shift value (13 for ROT13 and a variable for Caesar). Since the ESP32 executes nearly identical instructions in both cases, the time taken and the number of cycles consumed are the same.

This similarity in performance is less noticeable on the ATMEGA328P due to differences in processing speed and architecture between the 8-bit ATMEGA328P and the 32-bit ESP32. However, on the faster and more powerful ESP32, these lightweight operations (shifting letters) run so efficiently that the time difference becomes negligible.

Both the Caesar and ROT13 methods can be mathematically described as:

For a letter L in the alphabet, where $L \in \{A, B, \dots, Z\}$, the Caesar cipher shifts L by a key k:

Equation 6.2: Caesar Shifting

$$L' = (L + K)\ mod\ 26$$

For ROT13, the shift k is always fixed to 13:

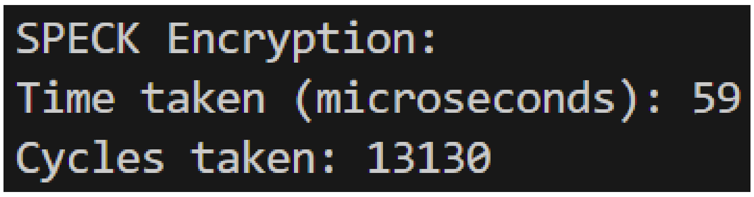
Equation 6.3: ROT13 Shifting

$$L' = (L + 13)\ mod\ 26$$

In both cases, the primary operation is the modular arithmetic involved in shifting the letters, which takes the same number of computational cycles for both Caesar and ROT13 because the difference lies only in the value of k, not in the complexity of the operations. Thus, on the ESP32, the Caesar cipher with a shift of 13 is identical to ROT13 in terms of execution, which explains why they show the exact cycle count and execution time.

6.1.4 Cycle Testing SPECK Encryption

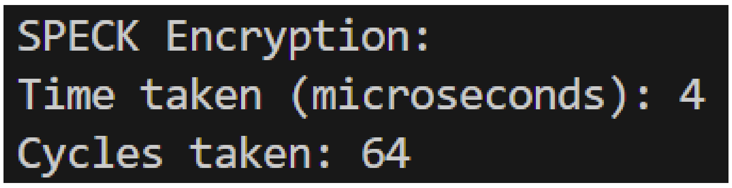
ESP32-S3 SPECK cycle encryption testing shows that the time taken in microseconds is 59µS, and 13130 cycles are required to complete this; please see *Figure 6.9*.



```
SPECK Encryption:
Time taken (microseconds): 59
Cycles taken: 13130
```

Figure 6.9: ESP32-S3 SPECK cycle testing serial print output

In comparison, the ATMEGA328P SPECK cycle encryption testing shows that the time taken in microseconds is 4µS and 64 cycles to complete this; please see *Figure 6.10*.



```
SPECK Encryption:
Time taken (microseconds): 4
Cycles taken: 64
```

Figure 6.10: ATMEGA328P SPECK cycle testing serial print output

There is a discrepancy between the time taken for the SPECK encryption between the ESP32 and the ATMEGA328P, which could indeed be linked to processing limitations or errors on the ATMEGA328P. The ESP32 is a 32-bit microcontroller that operates at higher clock speeds (up to 240 MHz). Its larger register size and faster clock rate give it the ability to handle more complex operations, like encryption algorithms, much more efficiently. The ATMEGA328P, on the other hand, is an 8-bit microcontroller typically operating at around 16 MHz. It has a smaller register size and a slower clock rate, which means that operations involving larger data sets (such as encryption) require significantly more cycles to complete.

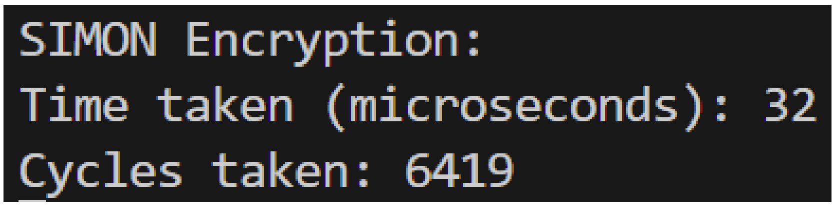
The SPECK algorithm is designed to be lightweight, but it still involves a sequence of bit shifts, additions, and XOR operations on 32-bit blocks. For a 32-bit microcontroller (like the ESP32), such operations can be executed natively without breaking them down into smaller chunks. On the ATMEGA328P, however, these 32-bit operations need to be broken down into multiple smaller 8-bit operations. This results in a much higher number of cycles for each operation, even though the overall complexity of the algorithm remains the same.

Given the significantly lower clock speed and reduced computational power of the ATMEGA328P, the algorithm might not be running optimally. It’s possible that the encryption could be interrupted or mismanaged due to the lack of processing resources, leading to a result that seems artificially "faster" because the operation is incomplete or incorrectly executed. This could explain why the ATMEGA328P result shows a much shorter time (59 microseconds), even though it would be expected to take much longer based on its processing power.

For the ESP32, the measured cycles (13130) reflect the actual time and complexity of the encryption process. This number of cycles seems appropriate for a lightweight encryption algorithm running on a 32-bit platform at a high clock speed. On the ATMEGA328P, the reported cycles (64) suggest a significant issue in either measurement or execution. Given the reduced processing power, the cycles should be much higher. This points to a potential problem with how the encryption algorithm is being handled by the microcontroller, likely related to either resource exhaustion or inaccurate measurement. Therefore, due to its complexity, there is likely an issue with the ATMEGA328P's handling of the SPECK encryption. The SPECK algorithm was likely either prematurely interrupted or incorrectly executed, leading to a cycle count and time taken that are far too low for the algorithm's actual complexity. This could indicate that the ATMEGA328P might not be suitable for handling such encryption tasks without significant optimisation or simplified alternatives.

6.1.4 Cycle Testing SIMON Encryption

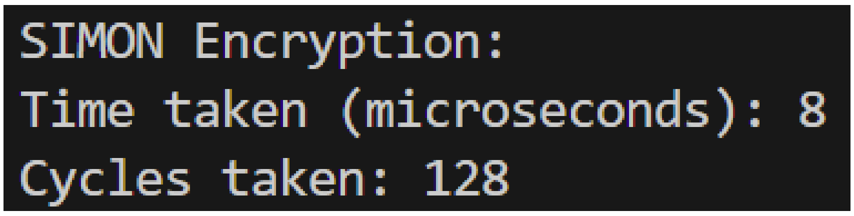
ESP32-S3 SIMON cycle encryption testing shows that the time taken in microseconds is 32µS, and 6419 cycles are required to complete this; please see *Figure 6.11*.



```
SIMON Encryption:
Time taken (microseconds): 32
Cycles taken: 6419
```

Figure 6.11: ESP32-S3 SIMON cycle testing serial print output

In comparison, the ATMEGA328P SPECK cycle encryption testing shows that the time taken in microseconds is 8µS and 128 cycles to complete this; please see *Figure 6.12*.



```
SIMON Encryption:
Time taken (microseconds): 8
Cycles taken: 128
```

Figure 6.12: ATMEGA328P SIMON cycle testing serial print output

The ATMEGA328P's lower cycle count could suggest that it is not running the full encryption algorithm. The low time (8 µs) and cycle count (128) compared to the ESP32-S3 (32 µs and 6419 cycles) implies that the ATMEGA328P may be prematurely finishing its encryption process, possibly due to an issue with how it is handling the SIMON algorithm. The ATMEGA328P is an 8-bit microcontroller with a 16-bit address space, while the ESP32-S3 is a 32-bit microcontroller. In general, a 32-bit processor like the ESP32-S3 should be able to handle larger blocks of data (e.g., 32-bit blocks in encryption) more efficiently, as it is natively optimised for such operations. However, the ATMEGA328P might be facing challenges in managing this, leading to unexpected completion of the algorithm without performing all intended steps. The ATMEGA328P has a relatively lower clock frequency (16 MHz) compared to the ESP32-S3 (up to 240 MHz), and while both systems should report cycle counts based on their respective CPU speeds, the large gap suggests either incorrect cycle measurement on the ATMEGA328P or a logic error in how the encryption process is executed.

SIMON and SPECK are both lightweight ciphers, but they still require multiple rounds of bitwise operations. A misinterpretation or early termination of those rounds on the ATMEGA328P might result in artificially low cycle counts. The ATMEGA328P uses an AVR architecture with fewer instructions per cycle compared to the more powerful Xtensa architecture on the ESP32-S3, which could result in the ATMEGA328P skipping or simplifying operations.

6.2 Detailed Analysis and Interpretation of Adaptive Amoeba Battery Curve Mapping Management System Performance

This sub-chapter uses the power analysis tool Otii to see the current, voltage, and power consumption of each encryption method tested in the previous sub-chapter (XOR, Caesar, ROT13, SPECK, and SIMON). The code remains the same as before, but with longer delays within the loops, making it easier to analyse the different selections, as shown in *Figure 6.13*.

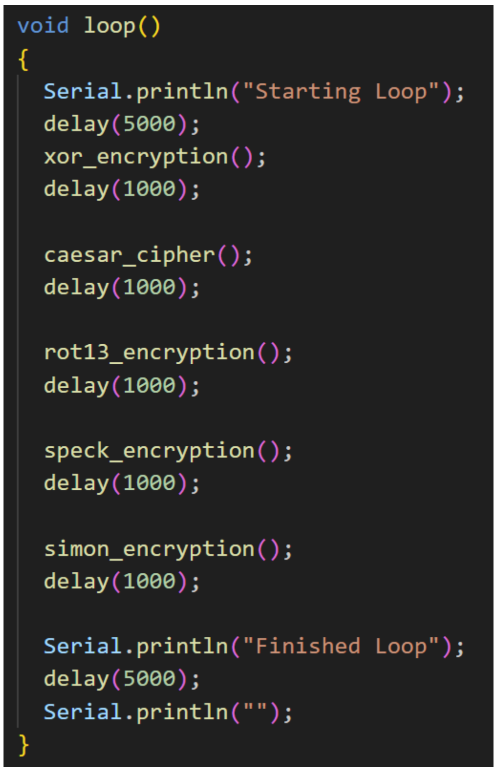

```
void loop()
{
  Serial.println("Starting Loop");
  delay(5000);
  xor_encryption();
  delay(1000);

  caesar_cipher();
  delay(1000);

  rot13_encryption();
  delay(1000);

  speck_encryption();
  delay(1000);

  simon_encryption();
  delay(1000);

  Serial.println("Finished Loop");
  delay(5000);
  Serial.println("");
}
```

Figure 6.13: Modified code for the encryption cycle testing for easy selection picking per method

The data collected using the Otti power analysis toolkit provides compelling insights into the performance and energy efficiency of various encryption schemes implemented as part of the Bio-

Inspired Lightweight Polymorphic Security System for IoT Devices. This analysis presents a detailed comparative analysis of encryption methods implemented within the Bio-Inspired Lightweight Polymorphic Security System, focusing on their execution on two microcontrollers: the ESP32-S3 (32-bit) and the ATMEGA328P (8-bit). The analysis evaluates time efficiency, cycle consumption, power draw, and energy usage. These factors are critical in understanding how the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS) dynamically optimises energy usage in resource-constrained environments. Please see the table below for the comparative results. This performance is benchmarked quantitatively against standard lightweight ciphers as detailed in *Table 7*. See Table 7 and Appendix 17 for full quantitative benchmarking against standard lightweight encryption schemes.

Table 7: Encryption Performance Comparison

Encryption Power Performance Summary										
Encryption Method	**ESP32 Time (µS)**	**ESP32 Cycles**	**ESP32 Avg Power (mW)**	**ESP32 Avg Current (mA)**	**ESP32 Energy (µJ)**	**ATMEGA328P Time (µS)**	**ATMEGA328 Cycles**	**ATMEGA328P Avg Power (mW)**	**ATMEGA328P Avg Current (mA)**	**ATMEGA328P Energy (µJ)**
Idel (Control)	N/A	N/A	227	49.7		N/A	N/A	8.12	2.17	
XOR	3	514	230	50.6	690	60	960	11	2.93	660
Caesar	17	3941	231	50.6	3927	728	11648	10.9	2.92	7935
ROT13	17	3941	231	50.8	3927	724	11584	10.8	2.89	7819
SPECK	51	12183	232	50.9	11832	4	64	11	2.92	44
SIMON	19	4515	231	50.8	4389	8	128	10.9	2.91	87

The ESP32 consistently demonstrates faster execution times and higher energy demands across most encryption tasks. For example, the XOR encryption method completes in 3µs on the ESP32 (514 cycles), whereas the same operation takes 60µs on the ATMEGA328P (960 cycles). However, due to the ESP32's higher operating frequency and current draw (~230 mW on average), the energy consumption for XOR is 690µJ on the ESP32 versus just 660µJ on the ATMEGA328P.

In contrast, block cipher methods such as SPECK highlight a compelling anomaly. Despite the ESP32 taking 51µs (12,183 cycles), the ATMEGA328P records just 4µs (64 cycles), which is likely an artefact of under-measured execution due to the ATMEGA's limited cycle timer resolution or optimisation artefacts. This discrepancy results in a dramatic difference in calculated energy: 11,832µJ on the ESP32 versus only 44µJ on the ATMEGA328P. Such outliers underscore the need for methodological caution when interpreting low-level timing on constrained hardware.

The ROT13 and Caesar methods yield identical performance on the ESP32 (17µs and 3,941 cycles each), which is expected because ROT13 is mathematically a Caesar cipher with a fixed shift of 13 positions. This redundancy in implementation explains the matching cycle counts and energy costs (3,927µJ each). On the ATMEGA328P, ROT13 and Caesar also exhibit close timing and energy profiles (728µs/11648 cycles vs. 724µs/11584 cycles), affirming functional parity across platforms.

Time (t) is a direct metric of responsiveness. ESP32, operating at a significantly higher frequency (~240 MHz vs. 16 MHz), exhibits lower latencies across all methods. This supports its suitability for latency-sensitive tasks, but at the expense of higher instantaneous power draw. ATMEGA328P, while slower, is advantageous in ultra-low-power scenarios due to its lower baseline power consumption.

Cycle count (C) reflects instruction throughput:

Equation 6.4: Cycle Count for Instruction Throughput

$$C = t \times f$$

where f is the CPU clock frequency, the proportionality confirms measurement integrity except in the SPECK and SIMON tests on the ATMEGA328P, where exceptionally low cycle counts suggest misaligned measurement windows or computational pipeline misreads. These anomalies merit repeat testing using oscilloscope-triggered time markers or more granular profiling tools.

Power (P) in milliwatts and current (IIn milliamps, it directly affects battery drain. Despite the ESP32's efficiency in reducing time, it operates at nearly 20x the power draw of the ATMEGA328P. This necessitates intelligent sleep-wake cycles to avoid rapid battery depletion, precisely what the AABCMS is designed to handle.

Energy (E) is calculated by:

Equation 6.5: General Formulation for Calculating Energy

$$E = P \times t$$

Where P is in milliwatts and t in microseconds (converted appropriately to seconds), energy consumption directly quantifies the operational cost of a computation. The ESP32's high power draw offsets its shorter execution time, whereas the ATMEGA328P achieves energy frugality through slower but less power-intensive operations.

The AABCMS is designed to respond dynamically to battery voltage and computational demand. Using sensor data, it adjusts operational modes (normal, light sleep, deep sleep, or ULP) based on thresholds derived from power usage. For instance, during Caesar and ROT13 operations, if battery health is $\geqslant$ 50%, the system may opt for light sleep post-processing due to moderate energy usage. Conversely, after SPECK operations on the ESP32, which consume over 11 mJ per execution, the system may enforce deep sleep or longer sleep durations to prevent premature battery depletion.

This adaptive behaviour can be modelled by a threshold function:

Equation 6.6: Formulation for Adaptive Sleep Mode Functions

$$Sleep\ Mode = \begin{cases} Normal & if\ E < 1mJ \\ Light\ Sleep & if\ 1 \leq E < 4mJ \\ Deep\ Sleep & if\ 4 \leq E < 10mJ \\ ULP & if\ E \geq 10mJ \end{cases}$$

This logic enables dynamic scheduling and intelligent power management tailored to the computational and energy characteristics of each encryption task. The comparative testing reveals that while ESP32 delivers speed, it does so at a significant energy cost, especially for block cipher methods like SPECK and SIMON. The ATMEGA328P, though slower, is remarkably efficient in power-constrained environments. These findings validate the need for the AABCMS to dynamically adjust device behaviour based on encryption workload and battery metrics. Such adaptive mechanisms are not only effective in prolonging device lifespan but also critical in making cryptographically secure IoT systems viable in real-world, power-sensitive deployments.

6.3 Testing the Auto Detection System (ADS)

Initially, when implementing the ADS, the technique used MAC (Media Access Control) addresses, but was changed to UUID (Universally Unique Identifiers). The length and complexity of a UUID is 128 bits long and is represented as a 32-character hexadecimal string (8-4-4-4-12 format). There are 2^{128} possible UUIDs. Whereas a MAC address is 48 bits long and is represented as a 12-character hexadecimal string (6 bytes). There are 2^{48} possible MAC addresses. Therefore, mathematically, a much larger keyspace for UUID which increases security as UUID keyspaces $2^{128} \approx 3.4 \times 10^{38}$ whereas, the MAC keyspace is $2^{48} \approx 2.8 \times 10^{14}$ possible combinations. The larger the keyspace, the harder it is for an attacker to "brute-force" or guess the correct identifier.

For brute-forcing a MAC address, In this testing scenario, if an attacker can make one billion (10^9) guesses per second (1GHz Processing speed):

Equation 6.7: Formulation for Brute Forcing MAC at 1GHz

$$Time\ to\ brute\ force\ MAC = \frac{2^{48}}{10^9} \approx 280{,}000\ seconds\ \approx 3.24\ days$$

For brute-forcing a UUID, if an attacker can make one billion (10^9) guesses per second (1GHz Processing speed):

Equation 6.8: Formulation for Vrute Forcing UUID at 1GHz

$$Time\ to\ brute\ force\ UUID = \frac{2^{128}}{10^9} \approx 10^{29} seconds$$

This is an astronomically large number of seconds, far beyond the age of the universe, which is approximately 10^{17} seconds. Therefore, brute-forcing a UUID is practically impossible with today's technology.

From a high-level privacy and security view, MAC addresses are hardware identifiers that remain constant unless manually changed (MAC spoofing). They are often broadcast in network environments (such as Wi-Fi discovery) and can be used to track devices persistently. Attackers who obtain a MAC address may use it to identify a device in different environments. MAC addresses are primarily used in lower-level networking protocols like Ethernet and Wi-Fi. They are unique to a device's network interface but are not designed to provide security features or encryption.

UUIDs can be dynamically generated in software and change over time or during sessions, improving privacy. They are not tied to hardware and are not typically broadcast in the same way as MAC addresses. UUIDs can also be customised and scoped to different use cases, adding more flexibility for securing communications. UUIDs are widely used across various applications, including databases, APIs, and security systems, to provide unique identification across multiple platforms and use cases. They are more flexible for use in session management, security protocols, and environments where devices need to rotate identifiers to enhance security.

Therefore, the decision to use UUIDs instead of MAC addresses as the keyspace for UUIDs is exponentially larger than that for MAC addresses, making UUIDs much harder to guess or brute-force. UUIDs can be dynamically generated and are not tied to hardware, enhancing privacy. MAC addresses, on the other hand, are fixed for a device's network interface and can be used to track devices. UUIDs are better suited for session management in security systems, as they can be rotated or invalidated easily. UUIDs are more versatile and widely used in various software applications beyond just network communication. UUIDs provide stronger security, privacy, and flexibility compared to MAC addresses, making them a better choice for IoT systems where unique identification and session management are crucial.

The Auto Detection System (ADS) is the first line of defence in the Bio-Inspired Lightweight Polymorphic Security System for IoT devices, mirroring the innate immune system’s surveillance function. Much like how immune cells patrol the body, identifying pathogens via molecular patterns, the ADS continuously monitors incoming client connections to classify them as approved, unapproved, or desynced. This process is executed during the initial handshake between the client and server, where the client's UUID, processing capability, and session ID are verified.

From the server-side implementation:

```
if (isUUIDApproved(clientUUID)) {
if (!isSessionApproved(receivedSessionID, clientUUID)) {
...
}
}
```

This logic reflects the system’s internal whitelist (analogous to self-antigen recognition), where only clients pre-listed as approved UUIDs can proceed. If a UUID is unrecognised, the system classifies it as a foreign agent and invokes the auto-ejection process. Furthermore, if a UUID is valid but the session ID is invalid or missing, the system classifies the client as desynced.

Mathematically, this detection logic can be modelled as:

Equation 6.9: ADS Detection Logic

$$ADS_{decision} = \begin{cases} Approved \;\; if \; UUID \in \mathcal{A} \wedge SessionID_{valid} \\ Desynced \;\; if \; UUID \in \mathcal{A} \wedge SessionID_{invalid} \\ Unapproved \;\; if \; UUID \notin \mathcal{A} \end{cases}$$

Where $\mathcal{A}$ is the set of all pre-approved UUIDs. This system supports real-time surveillance and adapts dynamically as new clients attempt to connect, ensuring only legitimate agents are authorised into the ecosystem.

6.4 Testing the Auto Ejection System (AES)

The Auto Ejection System (AES) functions analogously to the adaptive immune system's cytotoxic response, which identifies and eliminates threats such as infected or malicious cells. Once a client is detected as unapproved or desynced, the AES actively terminates the session and purges the associated credentials. This ensures the integrity and security of the network by preventing further attempts from unauthorised sources.

In the server code:

```
if (!isUUIDApproved(clientUUID)) {
  client.println("Session ID blocked or compromised");
  client.stop();
}
```

And for session desynchronisation:

```
if (clientSSID != ledger[currentLedgerIndex].ssid || clientPassword !=
ledger[currentLedgerIndex].password) {
  client.println("Error: Desync detected. Please reconnect.");
  client.stop();
}
```

These snippets demonstrate that any deviation from the expected UUID list or ledger credentials triggers immediate rejection. The AES is crucial for maintaining zero tolerance toward rogue devices and serves as the enforcement mechanism for the ADS's classification decisions.

In formal terms, the ejection policy can be modelled as a binary response:

Equation 6.10: Ejection Policy Binary Response

$$E_{Client} = \begin{cases} 1 & if \; UUID \notin \mathcal{A} \vee Desync_{detected} \\ 0 & otherwise \end{cases}$$

Where E_{Client} implies forced disconnection. The system thus minimises the attack surface by ejecting any actor not conforming to security expectations.

6.5 Testing the Trigger System (Hormonal Response) and Shared Ledger (Blockchain) Change of Security Credentials

The Trigger and Shared Ledger System operates as a long-term immunological memory. Upon identification of a compromised client (e.g., a UUID attempting a connection with forged credentials or an expired session), the system does not merely eject the entity, it initiates a triggered response. This includes generating new credentials and broadcasting them to all approved clients via the Shared Ledger mechanism.

Implemented in the server code:

```
currentLedgerIndex = (currentLedgerIndex + 1) % 5;
updateServerCredentials();
notifyApprovedClients();
```

This logic reflects an active defence model, where detection of an anomaly triggers the propagation of new cryptographic materials to the network. Clients then re-authenticate using updated keys and credentials, minimising the potential fallout of compromise and realigning network security.

Let Ln represent the current ledger index. When a compromise is detected:

Equation 6.11: Ledger Indexing Logic

$$L_{n+1} = (L_n + 1)\ mod\ N$$

Where N is the total number of available ledger entries. This modulus-based rotation ensures circular cycling through pre-defined secure states. Clients that fail to re-align with the updated ledger are classified as desynchronised and re-enter the detection-ejection cycle.

Biologically, this process is analogous to clonal selection and immunological memory formation, where antigen exposure triggers an adaptive, systemic response that reshapes the immune repertoire and maintains long-term protection.

6.6 Full System Testing

The full system testing involves reviewing the system in detail, considering the client and server code provided, to see how it handles different scenarios, such as new, returning, and unapproved connections. I'll go through each scenario and identify any potential issues, then suggest and implement corrections as necessary. This section presents a comprehensive evaluation of the Bio-Inspired Lightweight Polymorphic Security System for IoT Devices under a variety of real-world operational scenarios. The goal of this chapter is to validate the system’s robustness, efficiency, and adaptability through scenario-based testing. Particular emphasis is placed on the integration of encryption streams, power-aware reconnections, session expiration, desynchronisation recovery, and the dynamic behaviour of the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS).

6.6.1 Scenario 1: New Connection

The Client-Side actions, firstly, start as the client attempts to connect to the server using stored ledger credentials. If unsuccessful, it tries to connect to the honeypot SSID as a fallback. Upon successfully connecting to the server, the client receives a session ID, encryption method, and the server's ledger details. The client reads sensor data, encrypts it using the provided encryption method, and sends it to the server.

On the server side, the server generates a session ID for the new connection. The server checks if the client's UUID is approved. If approved, it stores the session details and sends the ledger information to the client. The server decrypts and validates the sensor data received and checks whether all approved clients have reconnected, potentially stopping the honeypot if necessary.

Potential edge case issues: if the client fails to connect to the Wi-Fi (either with stored credentials or the honeypot), it might repeatedly attempt to reconnect, consuming significant resources. The server needs to manage the dynamic session expiration effectively to avoid prematurely ending a valid session.

When a new client connects to the server for the first time, the server initiates a secure handshake by generating a unique session ID, transmitting it to the client, and requesting identification details, including the UUID and device capabilities. Upon receiving these parameters, the server performs a UUID verification through the isUUIDApproved(uuid) function and, if approved, generates and stores a new session record. Please see *Figure 6.14*.

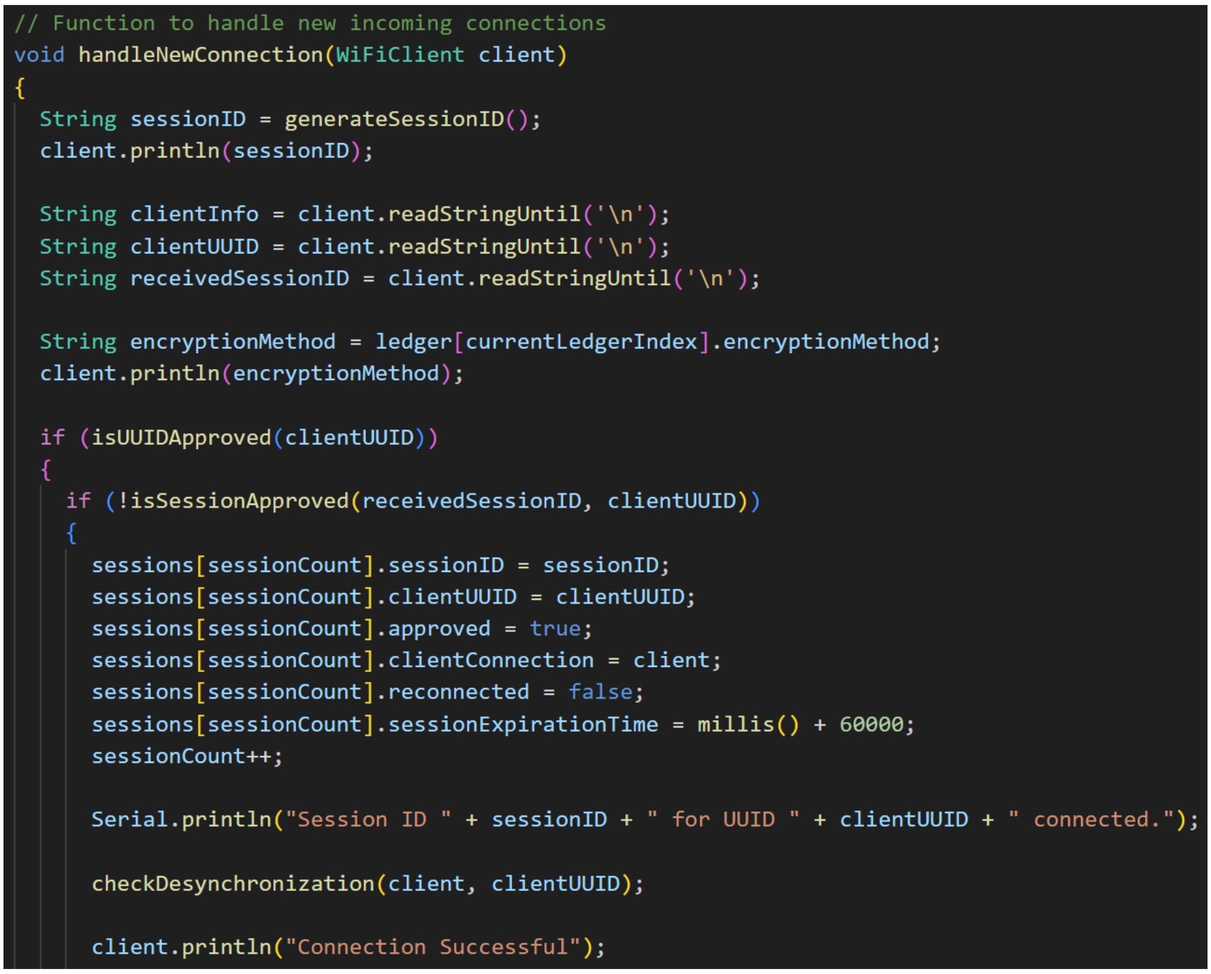

```
// Function to handle new incoming connections
void handleNewConnection(WiFiClient client)
{
  String sessionID = generateSessionID();
  client.println(sessionID);

  String clientInfo = client.readStringUntil('\n');
  String clientUUID = client.readStringUntil('\n');
  String receivedSessionID = client.readStringUntil('\n');

  String encryptionMethod = ledger[currentLedgerIndex].encryptionMethod;
  client.println(encryptionMethod);

  if (isUUIDApproved(clientUUID))
  {
    if (!isSessionApproved(receivedSessionID, clientUUID))
    {
      sessions[sessionCount].sessionID = sessionID;
      sessions[sessionCount].clientUUID = clientUUID;
      sessions[sessionCount].approved = true;
      sessions[sessionCount].clientConnection = client;
      sessions[sessionCount].reconnected = false;
      sessions[sessionCount].sessionExpirationTime = millis() + 60000;
      sessionCount++;

      Serial.println("Session ID " + sessionID + " for UUID " + clientUUID + " connected.");

      checkDesynchronization(client, clientUUID);

      client.println("Connection Successful");
```

Figure 6.14: Function for handleNewConnection on the server side to accept new connections.

It is essential to note the following lines of code:

> *String sessionID = generateSessionID();*
> *sessions[sessionCount].sessionID = sessionID;*
> *sessions[sessionCount].clientUUID = clientUUID;*

As a result, the client is granted access to the shared security ledger containing the current SSID, password, encryption method, and encryption key. The following code transmits these details securely:

> *client.println(ledger[currentLedgerIndex].ssid);*
> *client.println(ledger[currentLedgerIndex].password);*
> *client.println(ledger[currentLedgerIndex].encryptionKey);*
> *client.println(ledger[currentLedgerIndex].encryptionMethod);*

Subsequently, the client encrypts its sensor data string using the assigned encryption algorithm (e.g., XOR, Caesar, ROT13, SPECK, or SIMON) and transmits it to the server, which decrypts and logs the result. This transaction confirms a successful secure onboarding process.

Mathematically, this session handshake process is denoted by:

Equation 6.12: Session Handshake Process

$$SID = f(UUID, t)$$

Where SID is the dynamically generated session identifier, and tt is the timestamp, ensuring uniqueness per session.

6.6.2 Scenario 2: Returning Connection

On the client side, the actions are similar to a new connection; the client attempts to connect using stored credentials. Upon reconnecting, the client sends its session ID to the server to verify the connection. Returning clients use their stored session ID and ledger credentials to reconnect. If the session is still valid and the UUID is approved, the server allows the connection without reinitialisation.

The Server-Side actions first verify the session ID against stored sessions. If valid, the server continues the session without reinitialisation. The server updates the session's expiration time, keeping it active for another period.

Potential edge case issues could arise if the session expires while the client sleeps; the server might treat the client as a new connection, leading to unnecessary reinitialisation. The session expiration mechanism is sensitive to the client's sleep mode. If a client's session expires while sleeping, it may be marked as compromised. However, the system is designed to adapt by dynamically extending the session expiration time when the client enters sleep mode.

When a previously connected client reestablishes communication, it transmits its stored session ID and UUID. The server cross-verifies these via:

if (isSessionApproved(receivedSessionID, clientUUID))

If valid and not expired, the session is resumed without requiring full reinitialisation. The client is synchronised with the latest ledger credentials, unless there has been a recent security-triggered update. This mechanism greatly improves efficiency, reduces redundant transmissions, and supports device mobility and energy-aware sleep cycles.

The dynamic session validation logic incorporates timeouts via:

if (millis() > sessions[i].sessionExpirationTime)

This ensures that stale or inactive sessions are invalidated, maintaining security integrity across asynchronous reconnections.

6.6.3 Scenario 3: Unapproved Connection

The server rejects the connection if a client's UUID is not on the approved list. The session is flagged as compromised, and no further interaction occurs. A Potential Issue is that unapproved clients might attempt repeated connections, but the server is robust enough to detect and handle these attempts without disrupting approved clients. If a client attempts to connect with a UUID not found in the approved UUID list (*approvedUUIDs[]*), the system immediately blocks the session:

```
if (!isUUIDApproved(clientUUID)) {
  client.println("Session ID blocked or compromised");
  client.stop();
}
```

No credentials or ledger entries are transmitted, and the session is never created. This behaviour models a strict whitelist security architecture, essential for preventing intrusions or spoofing attacks. The UUID-based identification process is both cryptographically robust and dynamically verifiable.

The rejection logic is equivalent to the Boolean condition:

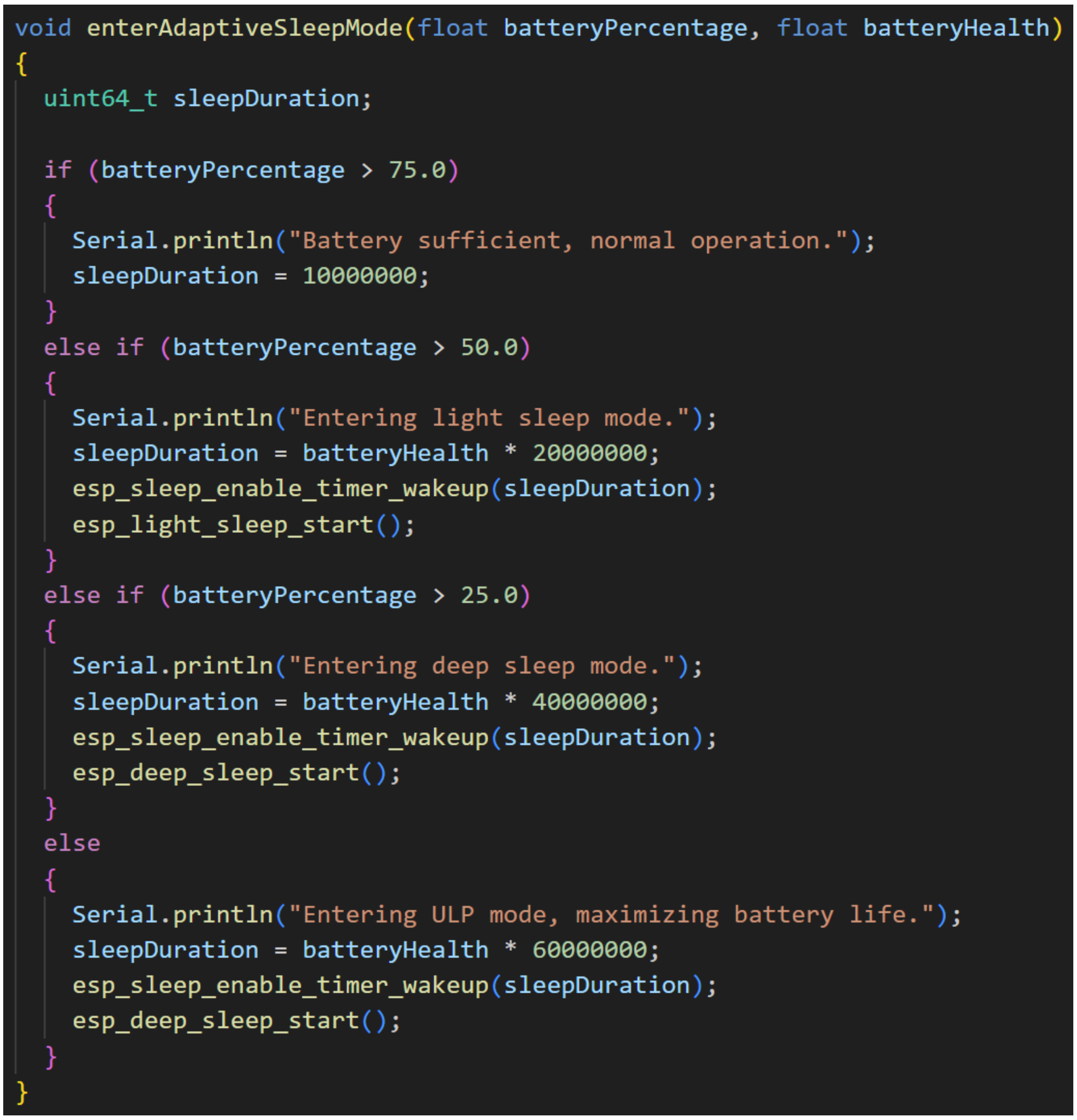
Equation 6.13: Rejection Logic for UUID

$$Access = \begin{cases} 1, & if\ UUID\ \in \mathcal{A} \\ 0, & otherwise \end{cases}$$

Where $\mathcal{A}$ Denotes the approved UUID set.

6.6.4 Scenario 4: Session Handling During Sleep Mode

In power-constrained conditions, the client enters light sleep, deep sleep, or ULP (Ultra Low Power) mode, as determined by the Adaptive Amoeba Battery Curve Mapping Management System. Depending on the predicted battery health and remaining energy, the system selects the optimal sleep strategy to conserve resources. Please see *Figure 6.15*.



```
void enterAdaptiveSleepMode(float batteryPercentage, float batteryHealth)
{
  uint64_t sleepDuration;

  if (batteryPercentage > 75.0)
  {
    Serial.println("Battery sufficient, normal operation.");
    sleepDuration = 10000000;
  }
  else if (batteryPercentage > 50.0)
  {
    Serial.println("Entering light sleep mode.");
    sleepDuration = batteryHealth * 20000000;
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_light_sleep_start();
  }
  else if (batteryPercentage > 25.0)
  {
    Serial.println("Entering deep sleep mode.");
    sleepDuration = batteryHealth * 40000000;
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_deep_sleep_start();
  }
  else
  {
    Serial.println("Entering ULP mode, maximizing battery life.");
    sleepDuration = batteryHealth * 60000000;
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_deep_sleep_start();
  }
}
```

Figure 6.15: Function for enterAdaptiveSleepMode on the client side

Upon waking, the client attempts to reconnect using its previously stored session ID. If the session has expired due to prolonged inactivity or timeouts on the server, the session is invalidated, and the client is prompted to reinitialise.

Additionally, to prevent premature disconnection of valid but sleeping clients, the server tracks the *isSleeping* flag per session and refrains from marking such sessions as expired:

```
if (sessions[i].approved && millis() > sessions[i].sessionExpirationTime
&& !sessions[i].isSleeping) {
  handleCompromisedCredentials(...);
}
```

This dynamic sleep-aware session management ensures maximum operational uptime and system resilience, especially for battery-powered IoT clients.

One of the critical challenges in energy-constrained environments, such as IoT deployments, is maintaining secure communication states while devices periodically enter low-power or sleep modes. Within the Bio-Inspired Lightweight Polymorphic Security System for IoT Devices, session persistence and re-authentication logic are intelligently coordinated with the device's battery health and operational context. This is particularly essential for platforms like the ESP32-S3, which utilise light sleep and deep sleep to prolong battery life while maintaining necessary network awareness and response readiness.

The Adaptive Amoeba Battery Curve Mapping Management System determines the optimal sleep mode for the device, ranging from normal operation to light, deep, or ultra-low-power (ULP) states, based on real-time voltage readings and battery health estimations. Upon entering sleep, the client device disconnects from Wi-Fi, potentially breaking its session with the server. To resolve this without compromising session integrity or requiring a full restart of communication logic, the framework maintains a persistent client profile through the shared ledger system and dynamic session management.

The key mechanism of session restoration is observed in the *connectToWiFiWithStoredLedger()* and *handleLedgerUpdate()* functions. These functions rely on the client's last valid credentials, including SSID, password, encryption key, and encryption method, as previously received from the server and stored locally. Upon waking from sleep, the device executes a routine that attempts to reconnect using the latest ledger entry.

```
if (connectToWiFiWithStoredLedger() && client.connect(host, port)) {
   Serial.println("Reconnected to server using ledger credentials");
}
else {
   Serial.println("Failed to reconnect to server");
   handleDesync();
}
```

In cases where the server has updated the ledger (e.g., due to a security breach or credential rotation) and the client's stored credentials are out of sync, the client is classified as desynced. The *handleDesync()* function then initiates a secure reconnection through a fallback "honeypot" SSID, allowing the client to present its UUID, validate itself as an approved device, and receive the updated ledger credentials:

```
client.println(uuid);  // Send UUID via honeypot
```

```
...
storedLedger[i].ssid = client.readStringUntil('\n');
storedLedger[i].password = client.readStringUntil('\n');
storedLedger[i].encryptionKey = client.readStringUntil('\n');
storedLedger[i].encryptionMethod = client.readStringUntil('\n');
```

This logic ensures that session recovery is autonomous and that security is not sacrificed for power conservation. Importantly, the system mimics biological resilience mechanisms, akin to immune memory and reactivation upon antigen exposure. The client behaves similarly to a memory B cell, retaining knowledge of past interactions and re-engaging upon re-exposure (wake-up), either by reasserting its identity (UUID and session ID) or by realigning itself through the updated security credentials (ledger rotation).

Mathematically, the reconnection process can be described as:

Equation 6.14: Reconnection Logic

$$R = \begin{cases} 1, & if\ (SSID, PWD, K, E)_{client} = (SSID, PWD, K, E)_{server} \\ 0, & otherwise \end{cases}$$

Where $R = 1$ implies successful reconnection. If $R = 0$, The desync pathway is initiated:

Equation 6.15: Desync Pathway Logic

$$Desync_{client} \rightarrow Honeypot_{SSID} \rightarrow UUID_{verification} \rightarrow Ledger_{refresh}$$

The strategic importance of this approach lies in its minimal disruption to ongoing operations. Even when the client enters a prolonged sleep mode to conserve energy, the system guarantees eventual reintegration into the secure communication network, assuming the client remains approved. This ensures that both energy and security domains are harmonised, fulfilling the core aim of this research: an adaptable, resilient, and lightweight polymorphic security system inspired by biological principles.

During extended low-power conditions, the Adaptive Encryption Engine plays a pivotal role in preserving communication functionality while minimising energy draw. In these instances, low-cost encryption methods (e.g., XOR, Caesar) are favoured, ensuring that lightweight data transactions may still occur even under severe energy constraints. As battery levels recover during periodic recharging or deep sleep recovery, higher-strength encryption is progressively reinstated without manual intervention.

6.6.5 Scenario 5: Desynchronisation and Honeypot Response

If a client reconnects using outdated credentials (SSID or password), desynchronisation is detected through:

```
if (clientSSID != ledger[currentLedgerIndex].ssid || clientPassword !=
ledger[currentLedgerIndex].password)
```

Upon detection, the client is rejected, and a desync message is issued:

```
client.println("Error: Desync detected. Please reconnect.");
client.stop();
```

Simultaneously, the server transitions to honeypot mode via:

```
WiFi.softAP("Honeypot_SSID", "Honeypot_Password");
```

Approved clients are notified of the updated credentials via the notifyApprovedClients() function, and the *startHoneypot()* mechanism ensures desynchronised or malicious clients are lured away without access to real data or operations.

6.6.6 Summary and Implications

The complete system testing affirms that the Bio-Inspired Lightweight Polymorphic Security System successfully detects, authenticates, encrypts, synchronises, and manages client connections dynamically. The incorporation of UUID-based authentication, dynamic session ID handling, real-time ledger propagation, and sleep-aware session preservation collectively enhances the resilience and security of IoT communications. The design directly reflects biological immune system principles such as memory, detection, rejection, and adaptation. The inclusion of the Adaptive Amoeba Battery Curve Mapping Management System ensures that all operations, including encryption and communication, are responsive to energy availability, making the system suitable for long-term deployment in constrained environments.

7.0 Chapter Seven: Conclusion, Reflection, Future Work

The preceding chapters have detailed the conceptualisation, design, implementation, and validation of a novel security framework tailored for Internet of Things (IoT) devices: the Bio-Inspired Lightweight Polymorphic Security System. Central to this work has been the translation of biological principles, particularly from immunology and neurobiology, into robust computational mechanisms that address critical challenges in IoT security and energy management.

This final chapter consolidates the primary research contributions, reflects upon the broader implications of the work, and identifies prospective pathways for future development and enhancement. In doing so, it reaffirms the essential role that bio-inspiration, adaptive systems, and polymorphic methodologies will play in securing the next generation of embedded and distributed technologies.

7.1 Summary of Research Contributions

This research has successfully introduced a bespoke security framework designed for the stringent constraints of IoT environments, characterised by limited processing capabilities, volatile network conditions, and finite power supplies. The Bio-Inspired Lightweight Polymorphic Security System integrates three fundamental components:

Firstly, the Approved Clients List Process (ACLP), the Client Connection Logic (CCL), and the Ledger Process (LP) collectively form the core of a dynamic session and credential management system. These mechanisms enable the adaptive authentication, monitoring, and revocation of client access based on real-time behaviours, mirroring biological immune responses in identifying and neutralising foreign agents.

Secondly, this work proposes and implements the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS), an innovative subsystem inspired by the adaptive energy regulation observed in living organisms. This system dynamically monitors battery health, predicts energy reserves through real-time curve mapping, and adaptively alters device behaviour, including encryption complexity and connection intervals, to optimise longevity without sacrificing security.

Thirdly, the research presents empirical validation through rigorous testing on custom-built ESP32-S3 development boards and comparative analyses using an ATMEGA328P microcontroller. Comprehensive testing scenarios, encompassing approved, unapproved, and desynchronised clients, demonstrated the framework's resilience, adaptability, and lightweight operational footprint.

The originality of this thesis lies not only in the technical implementations but in the seamless integration of concepts across biology, cybersecurity, embedded engineering, and systems optimisation, offering a multidisciplinary approach to contemporary challenges. Moreover, this research contributes to the academic community by providing a scalable and adaptable framework that can be expanded for future advances in Artificial Intelligence (AI), Large Language Models (LLMs), and post-quantum cryptography domains.

7.2 Personal Reflection on the Research Journey

The genesis and evolution of this research have been shaped not merely by theoretical exploration but by lived experience and adaptive problem-solving in the face of both technical and personal challenges. Initially, the focus of this thesis was solely the conceptualisation and implementation of a bio-inspired polymorphic security system for IoT devices. However, the realities of hardware experimentation, prolonged field testing, and unforeseen system vulnerabilities revealed a deeper, more intricate requirement: the necessity of energy-adaptive intelligence as an inherent component of security design.

The development of the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS) emerged organically from these pressures. It was during the critical phases of hardware deployment and debugging, at a time when the researcher's health constraints imposed physical limitations on the research process, that a profound parallel between biological adaptation and technological resilience became apparent. Just as organisms under duress conserve energy and prioritise essential functions, so too must embedded systems under energy scarcity adaptively modulate their operations to sustain core security processes without risking total failure.

This realisation necessitated a pivot in the research framework, expanding the initial scope to integrate dynamic power management as a first-class citizen of the security architecture. The research thus evolved into an exercise not only in system security, but in system survival, drawing deeper from immunological and neurobiological principles than initially envisaged.

This research journey has further underscored the fundamentally interdisciplinary nature of addressing real-world technological challenges. While the foundations in embedded systems, cryptography, and network security provided a basis, progress was only possible through the integration of insights from biology, energy systems engineering, and even psychological models of adaptive behaviour. Consequently, this project has fostered a personal intellectual transformation: embracing complexity not as an obstacle to be simplified away, but as a living feature of resilient systems to be modelled, mirrored, and managed.

Moreover, the challenges encountered during periods of personal illness paradoxically became a critical source of innovation. Constraints on energy, cognition, and time mirrored the constraints facing IoT devices in the field, offering an authentic, embodied understanding of what it means to engineer for resilience. The resulting framework, bio-inspired, adaptive, polymorphic, and lightweight, is not merely a theoretical construct but a product of necessity forged through the real constraints and adversities experienced throughout the doctoral journey.

In reflection, this research stands as a testament to the principle that the best engineering, like the most enduring life forms, is shaped not in ideal conditions but through persistent adaptation under duress. It is hoped that this thesis not only contributes meaningfully to the academic fields of cybersecurity, IoT engineering, and bio-inspired computation but also serves as a living example of resilience in research practice itself.

7.3 Future Work

While the bio-inspired lightweight polymorphic security system for IoT devices, complemented by the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS), demonstrates substantial efficacy in addressing critical security and energy management challenges, several promising avenues for future research and development emerge.

First, a deeper integration of machine learning (ML) and artificial intelligence (AI) techniques presents an opportunity to enhance the system's adaptive capabilities. While the current framework relies on deterministic triggers such as session validation failures, desynchronisation events, and voltage thresholds, future iterations could employ reinforcement learning (RL) models to dynamically optimise sleep cycles, encryption method selection, and ledger updates based on predictive models of client behaviour and environmental conditions. For example, employing Q-Learning or Deep Q Networks (DQN) could allow devices to learn optimal security strategies in response to adversarial behaviour or fluctuating network health over time.

Secondly, the integration of the proposed security framework with Large Language Models (LLMs), such as transformer-based architectures, offers intriguing possibilities. LLMs could be utilised not only for anomaly detection based on traffic analysis and behaviour modelling, but also to assist in the autonomous negotiation of new security protocols between devices, fostering a form of emergent, decentralised "immune response" across heterogeneous IoT networks. Such LLM-driven adaptations could extend the principle of bio-inspired polymorphism beyond encryption variability to encompass real-time security policy evolution.

Furthermore, the looming reality of quantum computing necessitates a critical future enhancement: quantum-resilient encryption. While lightweight block ciphers such as SPECK and SIMON provide efficiency in current embedded contexts, they are unlikely to withstand attacks from quantum adversaries leveraging Shor's algorithm or Grover's algorithm. Therefore, extending the encryption suite to incorporate post-quantum cryptographic (PQC) primitives, such as Lattice-based encryption (e.g., Kyber, NTRU) or hash-based signature schemes (e.g., XMSS), will be essential. Here, adaptive battery management remains crucial, as PQC algorithms tend to impose substantial computational and memory overheads, which must be carefully managed within constrained IoT devices.

Additionally, expansion into multi-layered shared ledgers offers potential enhancements. While the current shared ledger operates as a distributed memory for synchronisation of security credentials, future systems could incorporate blockchain-like decentralisation with lightweight consensus protocols (e.g., Practical Byzantine Fault Tolerance (PBFT) or Tangle-based Directed Acyclic Graphs (DAGs)) to further harden the system against single-point failures and targeted attacks. Mathematically, ledger operations could be optimised to maintain $O(1)$ lookup times and $O(log\ n)$ update complexities, ensuring scalability even with increased device populations.

Another promising direction concerns the application of the security framework in bio-electronic systems and wearable medical IoT devices, where both energy constraints and security risks are amplified. In such contexts, drawing even closer analogies to biological systems, for example, modelling shared ledger updates after hormonal signalling cascades or implementing quorum sensing mechanisms for collective decision making among IoT clusters, could yield architectures that are both robust and organically scalable.

Finally, from an engineering perspective, a notable opportunity for enhancement lies in the integration of a Hardware Abstraction Layer (HAL) into the Bio-Inspired Lightweight Polymorphic Security System for IoT Devices. A HAL constitutes a critical intermediary software layer that standardises interaction between the application framework and the underlying hardware. Rather than relying directly on microcontroller-specific function calls, such as analogRead(), WiFi.begin(), or EEPROM-specific methods, the system would interface through abstracted functions that are implemented differently depending on the hardware platform.

For example, a generic function readBatteryVoltage() could internally map to analogRead() on an ESP32 platform, but to a different ADC reading mechanism on an ARM Cortex-M0 or RISC-V microcontroller. Similarly, connectToWiFi() could abstract away the differences in network stack initialisation across different wireless chipsets.

Mathematically, the coupling *C* between the software framework and the underlying hardware can be described as:

Equation 7.1: Software Application of HAL

$$C_{direct} > C_{HAL}$$

Where, C_{direct} represents direct coupling when hardware-specific code is embedded throughout the application, C_{HAL} represents the coupling once a HAL is implemented, and ideally $C_{HAL} \to 0$, achieving maximum portability.

Introducing a HAL would thus decrease system rigidity and facilitate seamless migration across diverse embedded platforms. This approach would also align with modern engineering practices in scalable IoT system design, where longevity, adaptability, and hardware-agnostic development are prioritised. Moreover, in the context of security-critical systems, a HAL allows for easier auditing, updating, and certification processes as hardware layers evolve.

Therefore, integrating a properly layered HAL constitutes an important refinement direction, strengthening the robustness and future resilience of the proposed security framework.

In summary, the bio-inspired lightweight polymorphic security system presented herein lays a powerful foundation. Future work extending its intelligence, resilience, and adaptability through ML/AI integration, quantum security readiness, decentralised ledgers, and further biologically-informed mechanisms offers a rich research agenda poised to advance the frontiers of both cybersecurity and embedded systems engineering.

7.4 Chapter Summary

This chapter has presented a comprehensive synthesis of the research findings, explored the broader implications of the work, and outlined significant directions for future advancement. Through the implementation of the bio-inspired lightweight polymorphic security system for IoT devices, supported by the Adaptive Amoeba Battery Curve Mapping Management System (AABCMS), this thesis demonstrates a novel and highly adaptive approach to securing resource-constrained environments.

The Future Work section identified key opportunities to enhance the system's intelligence and resilience by integrating machine learning, leveraging large language models for real-time adaptation, and preparing for the emerging threats posed by quantum computing. Furthermore, it proposed the expansion of the shared ledger architecture towards decentralised, biologically-inspired consensus models to strengthen synchronisation and trust within distributed IoT networks.

The personal reflection provided insight into the organic development of the AABCMS, highlighting how the research journey itself mirrored the adaptive and self-regulating principles central to both biological systems and the proposed security framework. The experience of overcoming technical and personal challenges reinforced the broader thesis theme: that flexibility, resilience, and constant adaptation are essential in both engineered and natural systems.

In uniting principles from embedded systems, cybersecurity, cryptography, biology, and immunology, this research makes a significant contribution to the interdisciplinary development of secure IoT infrastructures. The bio-inspired methodology offers a robust paradigm capable of evolving alongside the rapidly changing technological landscape, ensuring that IoT systems can remain secure, energy-efficient, and sustainable in the face of emerging threats.

This concluding chapter will finalise the thesis by reflecting on the core achievements, evaluating the limitations of the current framework, and offering overarching closing remarks regarding the impact and legacy of this research within the broader academic and engineering communities.

References

[1] X. Li, P. K. K. Loh, and F. Tan, "Mechanisms of polymorphic and metamorphic viruses," *Proceedings - 2011 European Intelligence and Security Informatics Conference, EISIC 2011*, pp. 149–154, 2011, doi: 10.1109/EISIC.2011.77.

[2] A. K. Abbas, A. H. Lichtman, and S. Pillai, *Basic Immunology: Functions and of the Immune System*. 2016.

[3] IHS, "The Internet of Things : a movement , not a market Start revolutionizing the competitive landscape," *IHS Markit*, p. 9, 2017.

[4] C. A. Janeway, P. Travers, M. Walport, and M. Shlomchik, *Immunobiology - The Immune System in Health and Disease*, 5th ed. New York: Garland Publishing Library, 2007.

[5] K. Murphy, C. Weaver, M. & Weaver, R. Geha, and L. Notarangelo, *Janeway's Immunobiology*, 9th ed. New York and London: Garland Science, 2017.

[6] J. Qin, Z. Bai, and Y. Bai, "Polymorphic algorithm of JavaScript code protection," *Proceedings - International Symposium on Computer Science and Computational Technology, ISCSCT 2008*, vol. 1, pp. 451–454, 2008, doi: 10.1109/ISCSCT.2008.48.

[7] Y. Yin, Y. Gan, H. Liu, and Y. Hu, "Design of the late-model key exchange algorithm based on the polymorphic cipher," *IEEM2010 - IEEE International Conference on Industrial Engineering and Engineering Management*, vol. 1, pp. 1509–1513, 2010, doi: 10.1109/IEEM.2010.5674154.

[8] B. Zhongying and Q. Jiancheng, "Webpage encryption based on polymorphic Javascript algorithm," *5th International Conference on Information Assurance and Security, IAS 2009*, vol. 1, pp. 327–330, 2009, doi: 10.1109/IAS.2009.39.

[9] G. C. Kessler, "An Overview of Cryptography," garykessler.net. Accessed: Nov. 03, 2020. [Online]. Available: https://www.garykessler.net/library/crypto.html

[10] Michael Chui, Markus Löffler, and Roger Roberts, "The Internet of Things | McKinsey," McKinsey. Accessed: Dec. 08, 2020. [Online]. Available: https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-internet-of-things

[11] BMBF, "Industrie 4.0 - BMBF," www.bmbf.de. Accessed: Dec. 08, 2020. [Online]. Available: https://www.bmbf.de/de/zukunftsprojekt-industrie-4-0-848.html

[12] U. Hansmann, L. Merk, M. S. Nicklous, and T. Stober, *Pervasive Computing Handbook*. 2001. doi: 10.1007/978-3-662-04318-9.

[13] L. F. Rahman, T. Ozcelebi, and J. Lukkien, "Understanding IoT Systems: A Life Cycle Approach," *Procedia Comput Sci*, vol. 130, pp. 1057–1062, 2018, doi: 10.1016/j.procs.2018.04.148.

[14] A. Morize and R. Pointereau, "The Positive Way CONNECTED DEVICE LIFE CYCLE : HOW DOES IT IMPACT THE VIABILITY OF IOT CONTACTS," Paris, 2020.

[15] A. Allcock and M. Leonard, "2019 Manufacturing Trends Report," 2019.

[16] Gilad David Maayan, “The IoT Rundown For 2020: Stats, Risks, and Solutions -- Security Today,” Security Today. Accessed: Dec. 15, 2020. [Online]. Available: https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx?m=1&Page=1

[17] H. Tankovska, “Global IoT end-user spending worldwide 2017-2025 ,” Statista. Accessed: Dec. 15, 2020. [Online]. Available: https://www.statista.com/statistics/976313/global-iot-market-size/

[18] P. Matthews, *The Concise Oxford Dictionary of Linguistics*, 2nd ed. Oxford University Press, 2014. doi: 10.1093/acref/9780199675128.001.0001.

[19] H. Delfs and H. Knebl, *Information Security and Cryptography Introduction to Cryptography*. 2015.

[20] A. Klein, *Stream ciphers*, vol. 9781447150. 2013. doi: 10.1007/978-1-4471-5079-4.

[21] M. Dworkin, “Recommendation for Block Cipher Modes of Operation,” *National Institute of Standards and Technology Special Publication 800-38A 2001 ED*, vol. X, no. December, pp. 1–23, 2005.

[22] W. Stallings, *Cryptography and Network Security Principles and Practice, Sixth Edition*, 6th ed. New Jersey: Pearson, 2014.

[23] A. W. Dent and C. J. Mitchell, *User’s Guide to Cryptography and Standards*. Artech House, 2005.

[24] Aakanksha Gaur and The Editors of Encyclopaedia Britannica, “polymorphism | Definition, Examples, & Facts | Britannica,” Encyclopaedia Britannica. Accessed: Jan. 19, 2021. [Online]. Available: https://www.britannica.com/science/polymorphism-biology

[25] D. Flower and J. Timmis, *In Silico Immunology*, 1st ed., vol. 1, no. 1. London: Springer, 2006.

[26] J. Koret and E. Bachaalany, *The Antivirus Hacker’s Handbook*, 1st ed. Indianapolis: Wiley, 2015.

[27] A. Church, “A formulation of the simple theory of types,” *Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, Jun. 1940, doi: 10.2307/2266170.

[28] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston, *Object-oriented analysis and design with applications, third edition*, 3rd ed., vol. 33, no. 5. New Jersey: Addison-Wesley, 2008. doi: 10.1145/1402521.1413138.

[29] D. M. Harland, M. W. Szyplewski, and J. B. Wainwright, “An alternative view of polymorphism,” *ACM SIGPLAN Notices*, vol. 20, no. 10, pp. 23–35, Oct. 1985, doi: 10.1145/382286.382377.

[30] B. C. Pierce, *Types and Programming Languages*, 1st ed. Cambridge: The MIT Press, 2002.

[31] A. Liu and P. Ning, “TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks,” *Proceedings - 2008 International Conference on Information Processing in Sensor Networks, IPSN 2008*, pp. 245–256, 2008, doi: 10.1109/IPSN.2008.47.

[32] M. Alioto, “Energy-Efficient Security for IoT Devices,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, pp. 2934–2944, Nov. 2017, Accessed: Apr. 27, 2025. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8279638

[33] Ernst Niedermeyer, Donald L. Schomer, and F. H. Lopes da Silva, *Electroencephalography: Basic Principles, Clinical Applications, and Related Fields*, no. 69. Lippincott Williams & Wilkins, 2011.

[34] M. D. Schwartz and T. S. Kilduff, “THE NEUROBIOLOGY OF SLEEP AND WAKEFULNESS,” *Psychiatr Clin North Am*, vol. 38, no. 4, p. 615, Dec. 2015, doi: 10.1016/J.PSC.2015.07.002.

[35] A. Dorri and R. Jurdak, “Tree-Chain: A Fast Lightweight Consensus Algorithm for IoT Applications,” *Proceedings - Conference on Local Computer Networks, LCN*, vol. 2020-Novem, pp. 369–372, 2020, doi: 10.1109/LCN48667.2020.9314831.

[36] E. R. Naru, H. Saini, and M. Sharma, “A recent review on lightweight cryptography in IoT,” *Proceedings of the International Conference on IoT in Social, Mobile, Analytics and Cloud, I-SMAC 2017*, pp. 887–890, 2017, doi: 10.1109/I-SMAC.2017.8058307.

[37] F. Rahman, M. Farmani, M. Tehranipoor, and Y. Jin, “Hardware-Assisted Cybersecurity for IoT Devices,” *Proceedings - 2017 18th International Workshop on Microprocessor and SOC Test, Security and Verification, MTV 2017*, pp. 51–56, 2018, doi: 10.1109/MTV.2017.16.

[38] Intel, “Intel® Software Guard Extensions.” Accessed: Mar. 06, 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html

[39] ARM, “TrustZone – Arm Developer.” Accessed: Mar. 06, 2021. [Online]. Available: https://developer.arm.com/ip-products/security-ip/trustzone

[40] M. Xue, C. Gu, W. Liu, S. Yu, and M. O’Neill, “Ten years of hardware Trojans: a survey from the attacker’s perspective,” *IET Comput Digit Tech*, vol. 14, no. 6, pp. 231–246, 2020, doi: 10.1049/iet-cdt.2020.0041.

[41] A. Khalid, S. McCarthy, M. O’Neill, and W. Liu, “Lattice-based Cryptography for IoT in A Quantum World: Are We Ready?,” *Proceedings - 2019 8th International Workshop on Advances in Sensors and Interfaces, IWASI 2019*, pp. 194–199, 2019, doi: 10.1109/IWASI.2019.8791343.

[42] J. Lim, Y. Kim, and C. Yoo, “ChainVeri : Blockchain-based Firmware Verification System for IoT environment,” pp. 1050–1056, 2018, doi: 10.1109/Cybermatics.

[43] H. Seo, J. Choi, H. Kem, T. Park, and H. Kim, “Pseudo random number generator and Hash function for embedded microprocessors,” *2014 IEEE World Forum on Internet of Things, WF-IoT 2014*, pp. 37–40, 2014, doi: 10.1109/WF-IoT.2014.6803113.

[44] V. A. Thakor, M. A. Razzaque, and M. R. A. Khandaker, “Lightweight Cryptography Algorithms for resource-constrained IoT devices: A Review, Comparison and Research Opportunities,” *IEEE Access*, pp. 1–17, 2021, doi: 10.1109/ACCESS.2021.3052867.

[45] M. O. Ojo, S. Giordano, G. Procissi, and I. N. Seitanidis, “A Review of Low-End, Middle-End, and High-End Iot Devices,” *IEEE Access*, vol. 6, pp. 70528–70554, 2018, doi: 10.1109/ACCESS.2018.2879615.

[46] H. Henderson, *Encyclopedia of computer science and technology*. Facts On File, 2009.

[47] "SAM D21 Arm Cortex-M0+ Microcontrollers - Microchip Technology | Mouser." Accessed: May 26, 2022. [Online]. Available: https://www.mouser.co.uk/new/microchip/microchip-technology-sam-d21-mcus/

[48] "RP2040 specifications – Raspberry Pi." Accessed: May 26, 2022. [Online]. Available: https://www.raspberrypi.com/products/rp2040/specifications/

[49] "STM32H747XI - High-performance and DSP with DP-FPU, Arm Cortex-M7 + Cortex-M4 MCU with 2MBytes of Flash memory, 1MB RAM, 480 MHz CPU, Art Accelerator, L1 cache, external memory interface, large set of peripherals, SMPS - STMicroelectronics." Accessed: May 26, 2022. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32h747xi.html

[50] "ATMEGA4809 | Microchip Technology." Accessed: May 26, 2022. [Online]. Available: https://www.microchip.com/en-us/product/ATMEGA4809

[51] "Wi-Fi & Bluetooth MCUs and AIoT Solutions I Espressif Systems." Accessed: May 26, 2022. [Online]. Available: https://www.espressif.com/

[52] "nRF52840 - Bluetooth 5.2 SoC - nordicsemi.com." Accessed: May 26, 2022. [Online]. Available: https://www.nordicsemi.com/Products/nRF52840

[53] "BCM58712." Accessed: May 26, 2022. [Online]. Available: https://www.broadcom.com/products/embedded-and-networking-processors/communications/bcm58712

[54] Mark Raskino, Brian Gammage, and Jackie Fenn, "Gartner's Hype Cycle Special Report for 2009," Gartner. Accessed: Apr. 20, 2021. [Online]. Available: https://www.gartner.com/en/documents/1108412/gartner-s-hype-cycle-special-report-for-2009

[55] T. Erl, Z. Mahmood, and R. Puttini, *Cloud Computing Concepts, Technology & Architecture*. UPPER SADDLE RIVER, NJ: PRENTICE HALL.

[56] Blesson Varghese, "History of the cloud | BCS," BCS. Accessed: Apr. 20, 2021. [Online]. Available: https://www.bcs.org/content-hub/history-of-the-cloud/

[57] P. Mell and T. Grance, "NIST SP 800-145, The NIST Definition of Cloud Computing," Gaithersburg, 2011.

[58] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, "APPLIED CRYPTOGRAPHY".

[59] "Cryptography Theory and Practice Fourth Edition".

[60] William. Stallings, *Cryptography and network security : principles and practice*, 7th ed. Pearson, 2017.

[61] J. Daemen and R. V. Rijmen, "The Rijndael Block Cipher", Accessed: Apr. 27, 2025. [Online]. Available: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf

[62] E. Ronen, C. O'flynn, A. Shamir, and A.-O. Weingarten, "IoT Goes Nuclear: Creating a ZigBee Chain Reaction".

[63] K. A. Mckay, L. B. Meltem, S. Turan, and N. Mouha, “Report on Lightweight Cryptography”, doi: 10.6028/NIST.IR.8114.

[64] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, “The SIMON and SPECK Families of Lightweight Block Ciphers,” *Cryptology ePrint Archive*, 2013, Accessed: Apr. 27, 2025. [Online]. Available: https://eprint.iacr.org/2013/404

[65] S. Banik, A. Bogdanov, and F. Regazzoni, “Exploring energy efficiency of lightweight block ciphers,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9566, pp. 178–194, 2016, doi: 10.1007/978-3-319-31301-6_10.

[66] Amrita, C. P. Ekwueme, I. H. Adam, and A. Dwivedi, “Lightweight Cryptography for Internet of Things: A Review,” *EAI Endorsed Transactions on Internet of Things*, vol. 10, 2024, doi: 10.4108/EETIOT.5565.

[67] J. Charles A Janeway, P. Travers, M. Walport, and M. J. Shlomchik, “Immunobiology,” *Immunobiology*, no. 14102, pp. 1–10, 2001, Accessed: Apr. 27, 2025. [Online]. Available: https://www.ncbi.nlm.nih.gov/books/NBK10757/

[68] L. Wang *et al.*, *Engineering Probiotic E . Coli With a Type III Secretion System for Targeted Delivery of Therapeutic VHH The Harvard community has made this*, 9th ed., vol. 4, no. 1. Elsevier, 2021. Accessed: Apr. 27, 2025. [Online]. Available: https://doi.org/10.1016/j.btre.2021.e00680

[69] P. J. . Delves and I. M. . Roitt, *Roitt’s essential immunology*, 12th ed. Wiley-Blackwell, 2017. Accessed: Apr. 27, 2025. [Online]. Available: https://www.wiley.com/en-gb/Roitt’s+Essential+Immunology%2C+12th+Edition-p-9781118232873

[70] Thomas J. Kindt, Richard A. Goldsby, Barbara A. Osborne, and Janis Kuby, *Kuby Immunology - Google Books*, 6th ed. W. H. Freeman, 2007. Accessed: Apr. 27, 2025. [Online]. Available: https://www.google.co.uk/books/edition/Kuby_Immunology/oOsFf2WfE5wC?hl=en

[71] Bruce Alberts *et al.*, *Molecular biology of the cell*, 6th ed., no. 5. Garland Science, 2014.

[72] Polly Matzinger, “TOLERANCE, DANGER, AND THE EXTENDED FAMILY*,” *Annu. Rev. lmmunol*, vol. 12, pp. 991–1045, 1994, Accessed: Apr. 27, 2025. [Online]. Available: www.annualreviews.org/aronline

[73] Kenneth. Murphy, Casey. Weaver, and Charles. Janeway, *Janeway’s immunobiology*, 9th ed. Garland Science, 2016.

[74] Michael Crosby (Google), Nachiappan (Yahoo), Pradan Pattanayak (Yahoo), Sanjeev Verma (Samsung Research America), and Vignesh Kalyanaraman (Fairchild Semiconductor), *BlockChain Technology: Beyond Bitcoin*, vol. 2. Berkeley, 2016. Accessed: Apr. 27, 2025. [Online]. Available: https://scet.berkeley.edu/wp-content/uploads/AIR-2016-Blockchain.pdf

[75] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” *White Paper*, Accessed: Apr. 27, 2025. [Online]. Available: www.bitcoin.org

[76] K. Christidis and M. Devetsikiotis, “Blockchains and Smart Contracts for the Internet of Things,” *IEEE Access*, vol. 4, pp. 2292–2303, 2016, doi: 10.1109/ACCESS.2016.2566339.

[77] E. Androulaki *et al.*, “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains,” *Proceedings of the 13th EuroSys Conference, EuroSys 2018*, vol. 2018-January, Apr. 2018, doi: 10.1145/3190508.3190538.

[78] Miguel Castro and Barbara Liskov, “Practical Byzantine Fault Tolerance,” in *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans: Proceedings of the Third Symposiumon Operating Systems Design and Implementation, Feb. 1999, pp. 173–186. Accessed: Apr. 27, 2025. [Online]. Available: https://pmg.csail.mit.edu/papers/osdi99.pdf

[79] J. Polastre, J. Hill, and D. Culler, “Versatile low power media access for wireless sensor networks,” *SenSys’04 - Proceedings of the Second International Conference on Embedded Networked Sensor Systems*, pp. 95–107, 2004, doi: 10.1145/1031495.1031508.

[80] Y. Xiao, V. K. Rayi, B. Sun, X. Du, F. Hu, and M. Galloway, “A survey of key management schemes in wireless sensor networks,” *Comput Commun*, vol. 30, no. 11–12, pp. 2314–2341, Sep. 2007, doi: 10.1016/J.COMCOM.2007.04.009.

[81] C. E. Shannon, “Communication in the Presence of Noise,” *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949, doi: 10.1109/JRPROC.1949.232969.

[82] A. Adamatzky, “Towards fungal computer,” *Interface Focus*, vol. 8, no. 6, Dec. 2018, doi: 10.1098/RSFS.2018.0029.

[83] M. Buffi *et al.*, “Electrical signaling in fungi: past and present challenges,” *FEMS Microbiol Rev*, vol. 49, 2025, doi: 10.1093/FEMSRE/FUAF009.

Appendix 0: Introduction

The appendices presented in this thesis provide comprehensive supplementary documentation to support the main body of research. They are intended to ensure the reproducibility, transparency, and practical applicability of the developed Bio-Inspired Lightweight Polymorphic Security System for IoT Devices. The appendices are systematically organised to include detailed hardware schematics, source code listings for both server and client implementations, mathematical models, full benchmarking data for encryption algorithms, and supporting technical tables. Each appendix is referenced at relevant points throughout the thesis, and its inclusion enables independent verification of the research findings, facilitates further development by the research community, and enhances the pedagogical value for engineers, students, and practitioners. This structure reflects a commitment to open scientific practice, ensuring that every critical aspect of the system—from design and implementation to real-world testing—can be independently assessed, extended, or adapted for related applications.

Appendix 1: Table of consumer IoT boards

IoT Development Boards														
Name	Board Manufacture	Chip Manufactu	Processor	Connectivity	Encryption Module	Algroithm Type	Flash	SRAM	EEPROM / SF	Max Frequenc	Power Consumptio	Digital I/(	Analogue I,	URL
Arduino MKR 1000 WiFi	Arduino	ARM	SAMD21 Cortex-M0+ 32bit	ATWINC1500 802.11 b/g/n 2.4 GHz	ATECC508	ECC508 CryptoAuthentication	256 KB	32 KB	No	48 MHz	4~20 mA	8	8	https://docs.arduino.cc/hardware/mkr-1000-wifi
Arduino MKR FOX 1200	Arduino	ARM	SAMD21 Cortex-M0+ 32bit	Microchip® Smart RF ATA8520	None	None	256 KB	32 KB	No	48 MHz	4~20 mA	8	8	https://docs.arduino.cc/hardware/mkr-fox-1200
Arduino MKR GSM 1400	Arduino	ARM	SAMD21 Cortex-M0+ 32bit	u-blox SARA-U201 3.75G UMTS/HSPA solution with GSM/(E)GPRS	ATECC508A	ECC508 CryptoAuthentication	256 KB	32 KB	No	48 MHz	4~20 mA	8	8	https://docs.arduino.cc/hardware/mkr-gsm-1400
Arduino MKR NB 1500	Arduino	ARM	SAMD21 Cortex-M0+ 32bit	u-blox NINA-W102 Wi-Fi 802.11b/g/n at 2.4 GHz ISM bandand Bluetooth v4.2	ATECC508A	ECC508 CryptoAuthentication	256 KB	32 KB	No	48 MHz	4~20 mA	8	8	https://docs.arduino.cc/hardware/mkr-nb-1500
Ardiino MKR Vidor 4000	Arduino	ARM	SAMD21 Cortex-M0+ 32bit and Intel® Cyclone® 10CL016	u-blox NINA-W102 Wi-Fi 802.11b/g/n at 2.4 GHz ISM bandand Bluetooth v4.2	ATECC508A	ECC508 CryptoAuthentication	256 KB and 2 MB	32 KB and 8 Mb	No	48 - 200 MHz	4~20 mA	8	8	https://docs.arduino.cc/hardware/mkr-vidor-4000
Arduino MKR WAN 1310	Arduino	ARM	SAMD21 Cortex-M0+ 32bit	Murata CMWX1ZZABZ	ATECC508A	ECC508 CryptoAuthentication	256 KB	32 KB	No	48 MHz	4~20 mA	8	8	https://docs.arduino.cc/hardware/mkr-wan-1310
Arduino MKR WiFi 1010	Arduino	ARM	SAMD21 Cortex-M0+ 32bit	u-blox NINA-W102 Wi-Fi 802.11b/g/n at 2.4 GHz ISM bandand Bluetooth v4.2	ATECC508A	ECC508 CryptoAuthentication	256 KB	32 KB	No	49 MHz	4~20 mA	8	8	https://docs.arduino.cc/hardware/mkr-wifi-1010
Arduino MKR WAN 1300	Arduino	ARM	SAMD21 Cortex-M0+ 32bit	Murata CMWX1ZZABZ	ATECC508A	ECC508 CryptoAuthentication	256KB	32 KB	No	48 MHz	4~20 mA	8	8	https://docs.arduino.cc/hardware/mkr-wan-1300
Arduino Nano 33 IoT	Arduino	ARM	SAMD21 Cortex-M0+ 32bit	u-blox NINA-W102 Wi-Fi 802.11b/g/n at 2.4 GHz ISM bandand Bluetooth v4.2	ATECC608A-MAHDA-T Crypto IC	ECC P256 (ECDH and ECDSA), SHA256, AES-GCM	1 MB	256 KB	No	48 MHz	4~20 mA	14	8	https://docs.arduino.cc/hardware/nano-33-iot
Arduino Nano 33 BLE	Arduino	ARM	nRF52840 Mbed	u-blox NINA-B306 Bluetooth® 5	None	None	1 MB	256 KB	No	64 MHz	4~20 mA	14	8	https://docs.arduino.cc/hardware/nano-33-ble
Arduino Nano 33 BLE Sense	Arduino	ARM	nRF52840 Mbed	u-blox NINA-B306 Bluetooth® 5	None	None	1 MB	256 KB	No	65 MHz	4~20 mA	14	8	https://docs.arduino.cc/hardware/nano-33-ble-sense
Arduino UNO WiFi Rev2	Arduino	Microchip	ATmega4809	u-blox NINA-W102 Wi-Fi 802.11b/g/n at 2.4 GHz ISM bandand Bluetooth v4.2	ATECC608A-MAHDA-T Crypto IC	ECC P256 (ECDH and ECDSA), SHA256, AES-GCM	48KB	6KB	256 bytes EEPROM	16 MHz	11.4 mA	14	6	https://docs.arduino.cc/hardware/uno-wifi-rev2
Arduino Nano RP2040 Connect	Arduino	ARM	Raspberry Pi RP2040	u-blox NINA-W102 Wi-Fi 802.11b/g/n at 2.4 GHz ISM bandand Bluetooth v4.2	ATECC608A-MAHDA-T Crypto IC	ECC P256 (ECDH and ECDSA), SHA256, AES-GCM	16 MB	264 KB	448 KB ROM	133 MHz	24.8 mA	20	8	https://docs.arduino.cc/hardware/nano-rp2040-connect
Portenta H7	Arduino	ARM	ST STM32H747XI	Murata 1DX dual WiFi and Bluetooth® 5.1	NXP SE050C2 Crypto	Common Criteria EAL 6+ and FIPS 140-2 certified security	2MB	1MB	No	480 MHz and 240 MHz	TBC	22	8	https://docs.arduino.cc/hardware/portenta-h7
Portenta H7 Lite Connected	Arduino	ARM	STM32H747XI dual Cortex®-M7+M4 32bit	Murata 1DX dual WiFi and Bluetooth® 5.1	NXP SE050C2 Crypto	Common Criteria EAL 6+ and FIPS 140-2 certified security	2MB	1MB	No	480 MHz and 240 MHz	TBC	22	8	https://docs.arduino.cc/hardware/portenta-h7-lite-connected
Portenta X8	Arduino	ARM	STM32H747XI dual Cortex®-M7+M4 32bit	Murata 1DX dual WiFi 802.11b/g/n 65 Mbps and Bluetooth 5.1 BR/EDR/LE	NXP SE050C2 Crypto	Common Criteria EAL 6+ and FIPS 140-2 certified security	16 GB eMMC	2 GB DDR4 DRAM	No	481 MHz and 240 MHz	TBC	22	8	https://docs.arduino.cc/hardware/portenta-x8
AVR-BLE Development Board	Microchip	Microchip	ATmega3208	RN4870 BLE	ATECC608A-MAHDA-T Crypto IC	ECC P256 (ECDH and ECDSA), SHA256, AES-GCM	32 KB	4 KB	EEPROM 256 KB	20 MHz	11 mA	16	N/A	https://www.microchip.com/en-us/solutions/internet-of-things/iot-development-kits/avr-ble-and-pic-ble-development-boards
AVR-IoT WA Development Board	Microchip	Microchip	ATmega4808	ATWINC1500 802.11 b/g/n 2.4 GHz	ATECC608A-MAHDA-T Crypto IC	ECC P256 (ECDH and ECDSA), SHA256, AES-GCM	48 KB	6 KB	EEPROM 256 KB	20 MHz	11.4 mA	16	N/A	https://www.microchip.com/en-us/development-tool/EV15R70A
AZ3166 IOT Developer Kit	ARM and Microsoft Azure	MXCHIP	EMW3166	Cortex-M4 MCU and RF chip of 802.11 b/g/n 2.4 GHz	Built-in	WEP, WPA/WPA2, PSK	1 MB	256 KB	EEPROM 256 KB	2.14 GHz	13.49 mA	N/A Connecting Finger	N/A Connecting Finger	https://www.amazon.co.uk/Plugable-Integrated-Microsoft-Arduino-Software/dp/B07646QWN9
ESP32-DevKitC	Espressif	Espressif	ESP32-WROVER-B	802.11 b/g/n (802.11n up to 150 Mbps) at 2.4 GHz and Bluetooth V4.2 BR/EDR and BLE	Built-in	Hardware encryption/decryption based on AES	4 MB	8 MB	No	80 upto 240 MHz	27~68 mA	24	10	https://components101.com/microcontrollers/esp32-devkitc
ESP32-DevKitM-1	Espressif	Espressif	ESP32-U4WD	802.11 b/g/n (802.11n up to 150 Mbps) at 2.4 GHz and Bluetooth V4.2 BR/EDR and BLE	Built-in	Hardware encryption/decryption based on AES	448 KB ROM	520 KB	SPI 4 MB	80 upto 120 MHz	20~34 mA	24	10	https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/user-guide-devkitm-1.html
ESP32-PICO-KIT V4	Espressif	Espressif	ESP32-PICO-D4	802.11 b/g/n (802.11n up to 150 Mbps) at 2.4 GHz and Bluetooth V4.2 BR/EDR and BLE	Built-in	Hardware encryption/decryption based on AES	Integrated SPI flash	4 MB	SPI 4 MB	80 MHz	80 mA	34	10	https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-pico-kit.html
ESP32-PICO-MINI-02	Espressif	Espressif	ESP32 embedded, Xtensa®dual-core 32-bit LX6 microprocesso	802.11 b/g/n (802.11n up to 150 Mbps) at 2.4 GHz and Bluetooth V4.2 BR/EDR and BLE	Built-in	Hardware encryption/decryption based on AES	8 MB	2 MB	No	80 upto 240 MHz	27~68 mA	27	18	https://www.espressif.com/sites/default/files/documentation/esp32-pico-mini-02_datasheet_en.pdf
ESP-WROVER-KIT	Espressif	Espressif	ESP32-WROVER-B	802.11 b/g/n (802.11n up to 150 Mbps) at 2.4 GHz and Bluetooth V4.2 BR/EDR and BLE	Built-in	Hardware encryption/decryption based on AES	4 MB	8 MB	No	80 upto 240 MHz	27~68 mA	24	16	https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-wrover-kit.html
Particle Argon IoT Development Board	Particle	Nordic Semiconductor	nRF52840	Espressif ESP32-D0WD 2.4G and Bluetooth 4.2 and BLE	ARM Trustzone CryptoCell - 310 Cryptographic and security module	ECC, SRP, SHA-1 and 2, AES, ChaCha20/Poly1305	1 MB	256 KB	SPI 4 MB	2.4 GHz	25.8 mA	20	6	https://store.particle.io/products/argon-kit
Particle Boron IoT Development Boards	Particle	Nordic Semiconductor	nRF52840	u-blox SARA R410 LTE modem and u-blox SARA U201 2G/3G modem	ARM Trustzone CryptoCell - 310 Cryptographic and security module	ECC, SRP, SHA-1 and 2, AES, ChaCha20/Poly1305	1 MB	256 KB	SPI 4 MB	2.4 GHz	25.8 mA	20	6	https://www.mouser.co.uk/new/particle/particle-boron-dev-board/
Raspberry Pi 3 Model B+	Raspberry Pi	Broadcom and ARM	Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC	2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE	None	None	-	1GB LPDDR2 SDRAM	-	1.4 GHz	2500 mA	26	N/A	https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/
Raspberry Pi 4	Raspberry Pi	Broadcom and ARM	BCM2711 and Quad core Cortex-A72 (ARM v8) 64-bit SoC	2.4 GHz and 5.0 GHz IEEE 802.11ac wireless, Bluetooth 5.0, BLE and Gigabit Ethernet	None	None	-	2GB, 4GB or 8GB LPDDR4-3200 SDRAM	-	1.5 GHz	3000 mA	26	N/A	https://www.raspberrypi.com/products/raspberry-pi-4-model-b/
Raspberry Pi Zero	Raspberry Pi	Broadcom	Broadcom BCM2835	802.11n, Bluetooth 4.1, BLE	None	None	-	512 MB	-	1 GHz	1200 mA	26	N/A	https://www.raspberrypi.com/products/raspberry-pi-zero/

Figure Appendix 1.1: Table of consumer IoT boards

Appendix 2: Custom Development Board

2.1 Circuit Diagram – Schematic

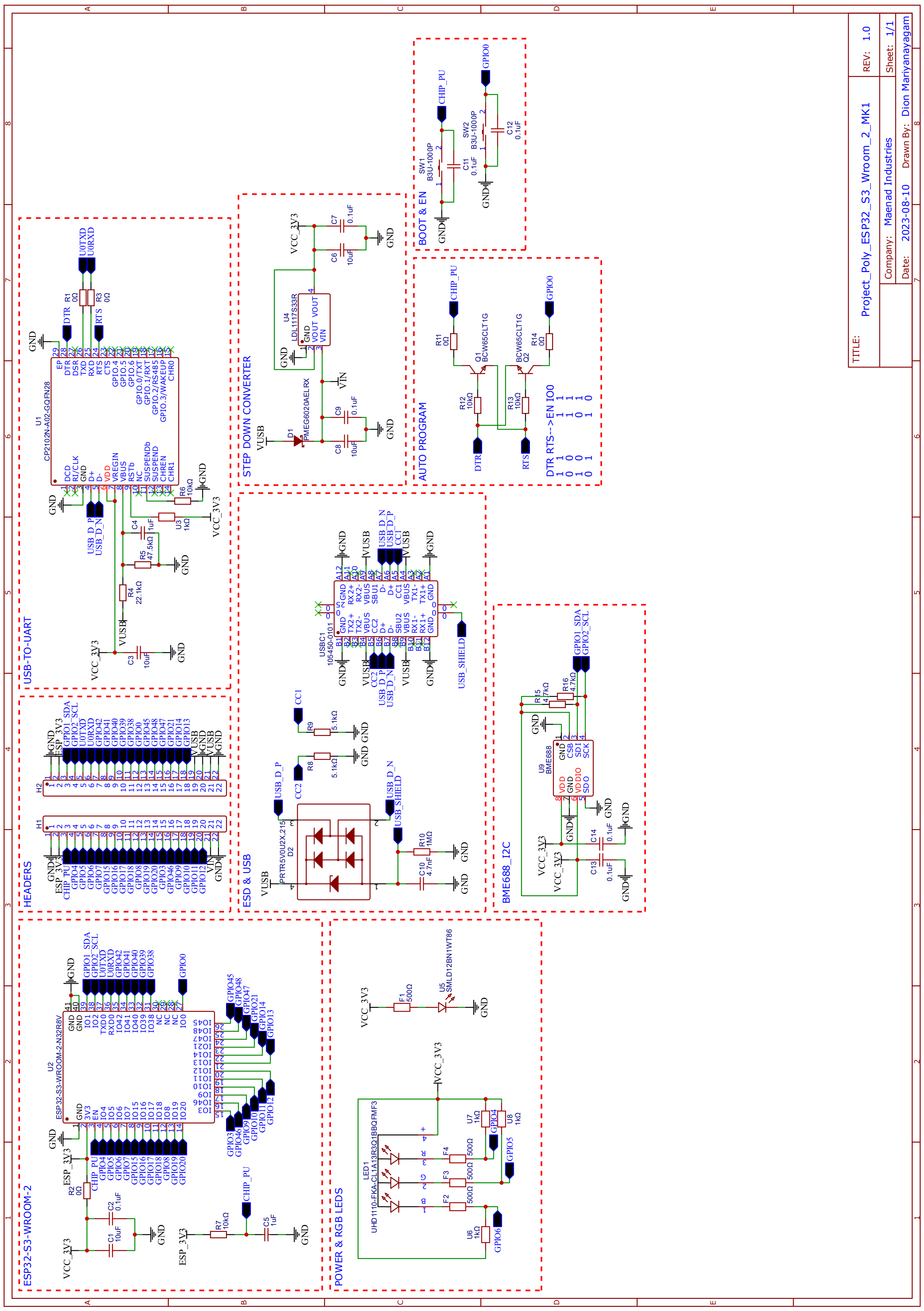


Figure Appendix 2.1: Circuit Diagram Schematic

2.2 PCB – Gerber Layers

2.2.1 Board Outline

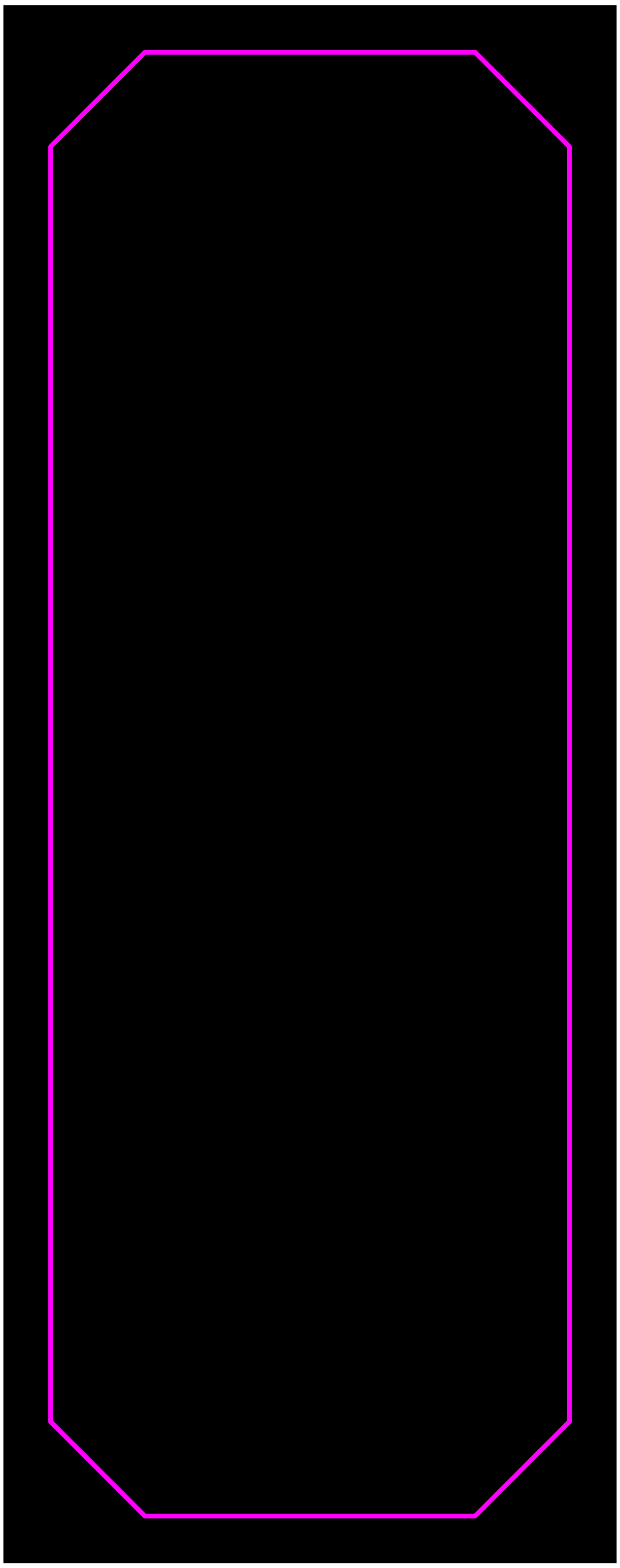


Figure Appendix 2.2: PCB Gerber Layer for the Board Outline

2.2.2 Bottom Layer

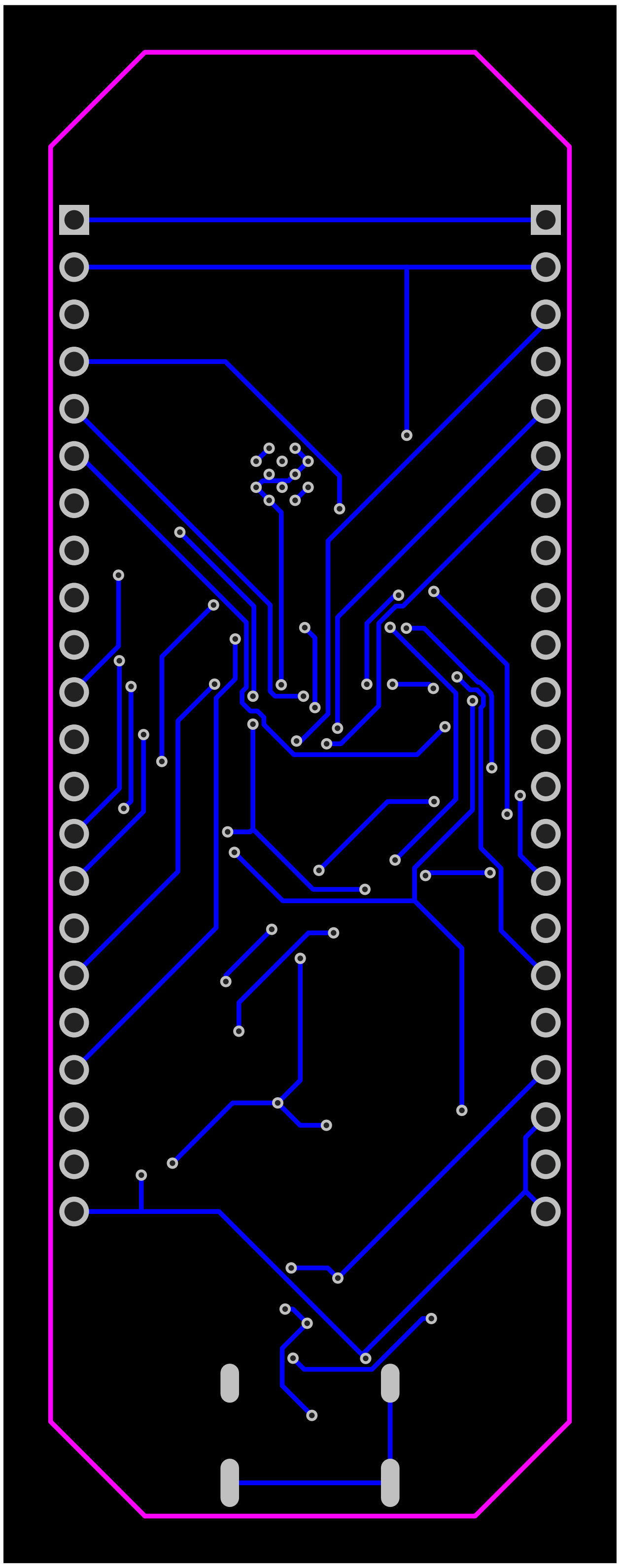


Figure Appendix 2.3: PCB Gerber layer for the Bottom Layer

2.2.3 Bottom Paste Mask Layer

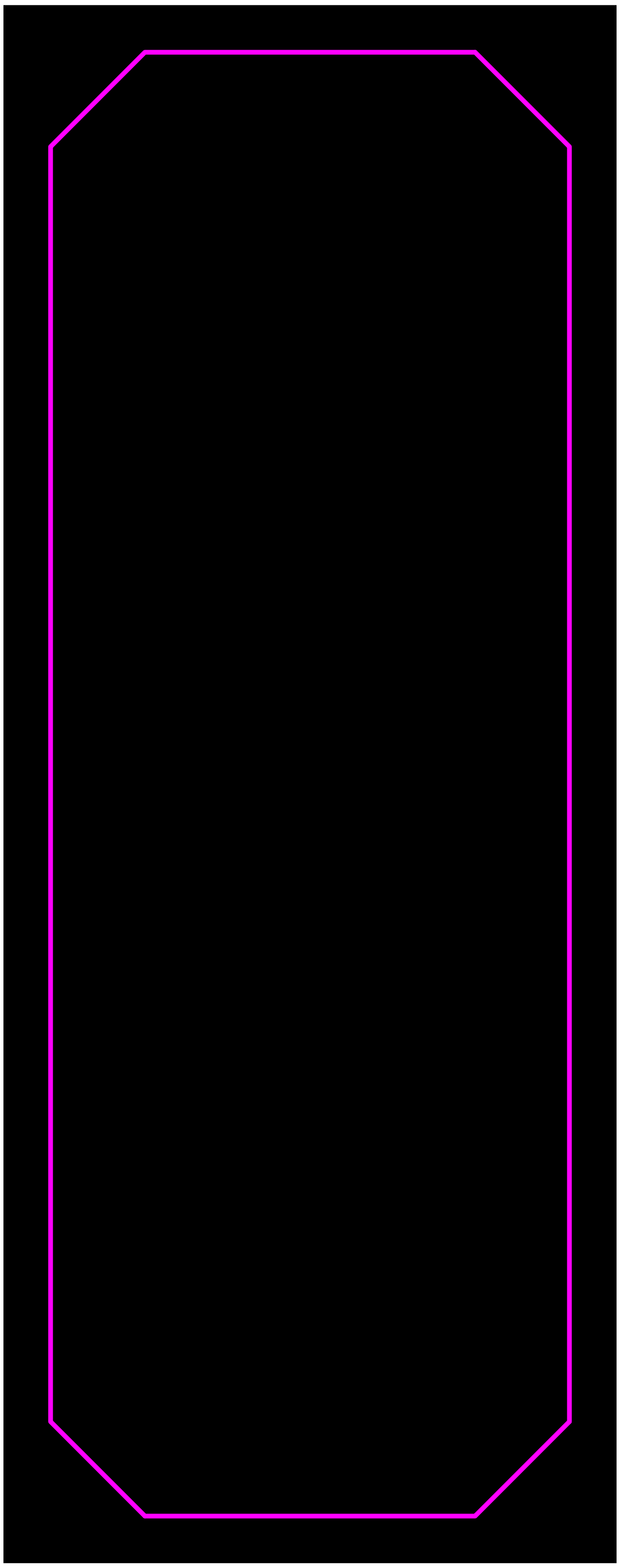


Figure Appendix 2.4: PCB Gerber Layer for the Bottom Paste Mask Layer

2.2.4 Bottom Silk Layer

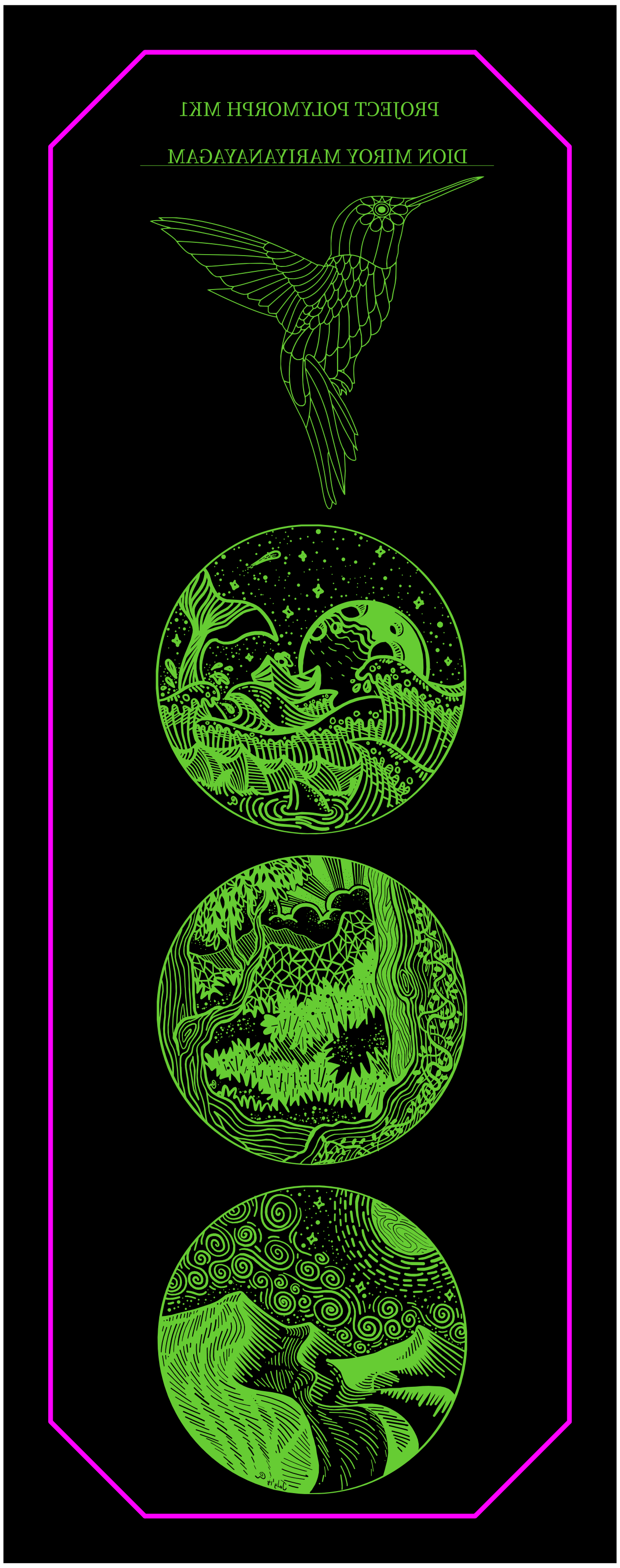


Figure Appendix 2.5: PCB Gerber Layer for the Bottom Silk Layer

2.2.5 Bottom Solder Mask Layer

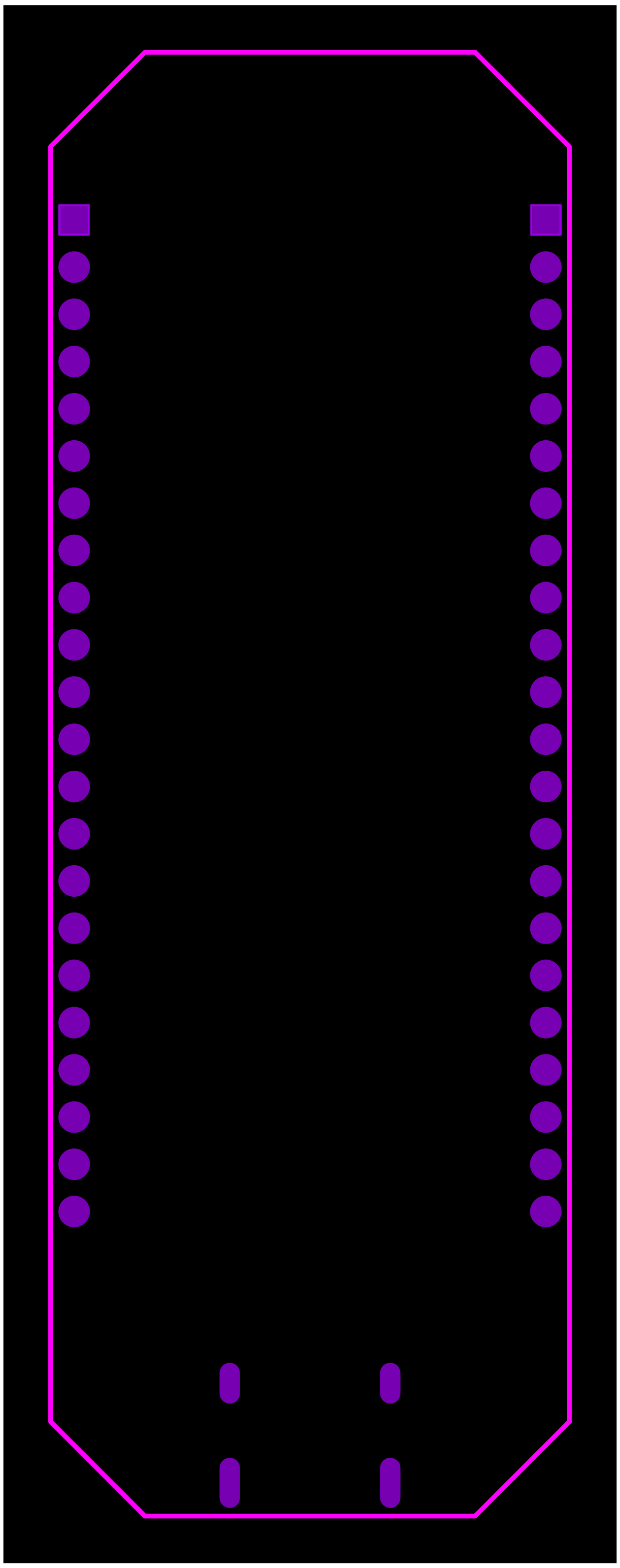


Figure Appendix 2.6: PCB Gerber Layer for the Bottom Solder Mask Layer

2.2.6 Document

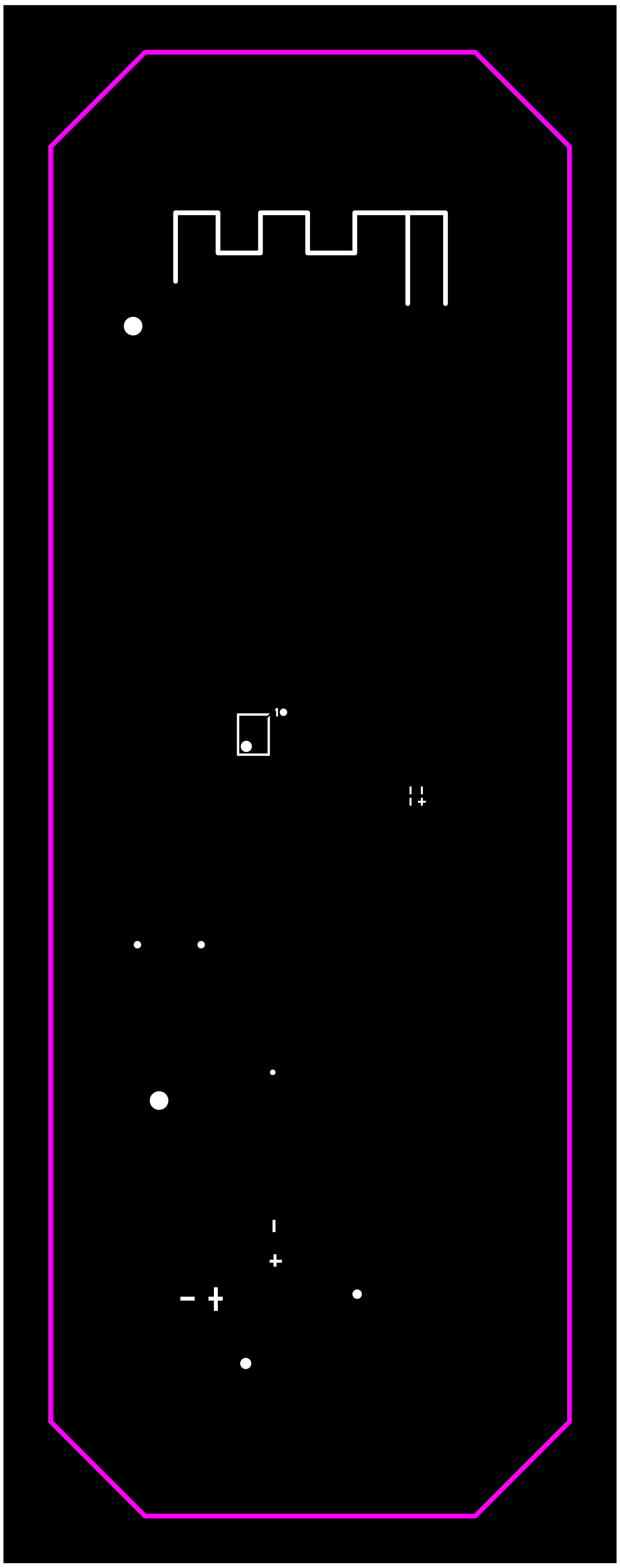


Figure Appendix 2.7: PCB Gerber Layer for the Document

2.2.7 Hole

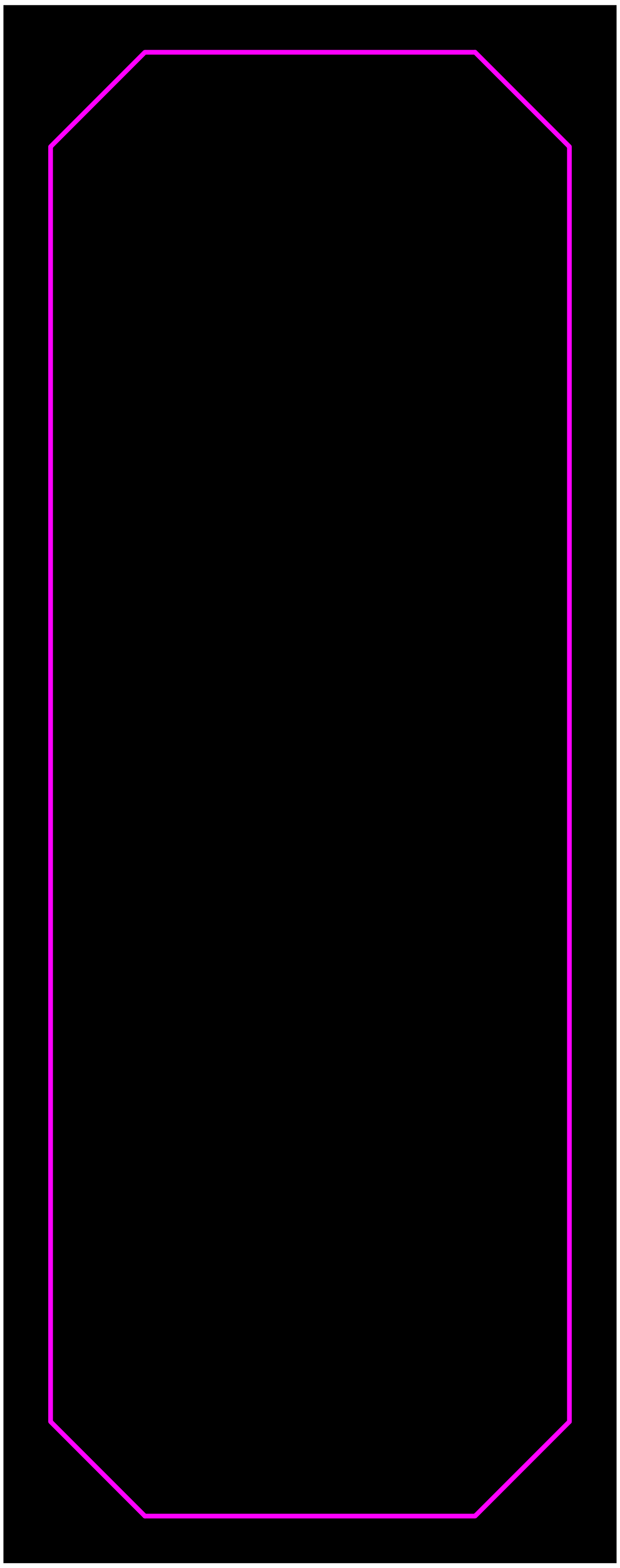


Figure Appendix 2.8: PCB Gerber Layer for the Hole

2.2.7 Inner1 - GND

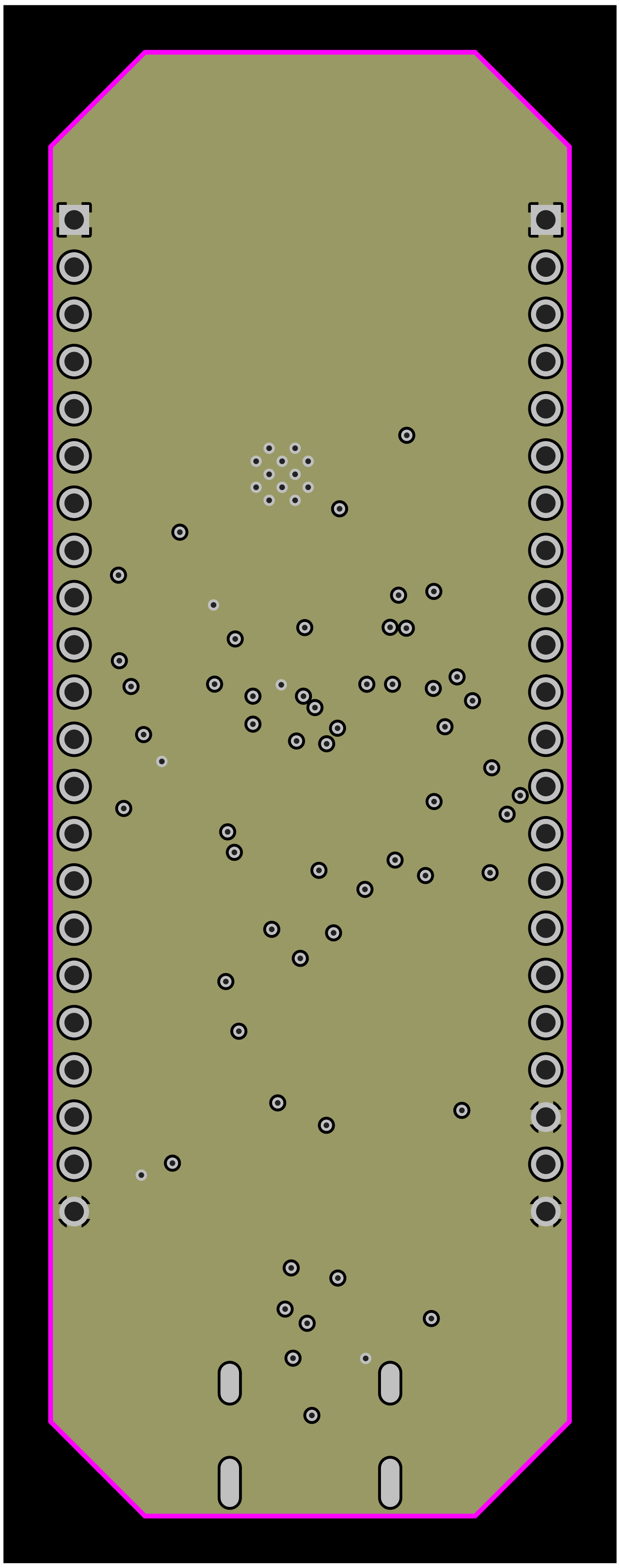


Figure Appendix 2.9: PCB Gerber Layer for the Inner1 - GND

2.2.8 Inner2 - VCC

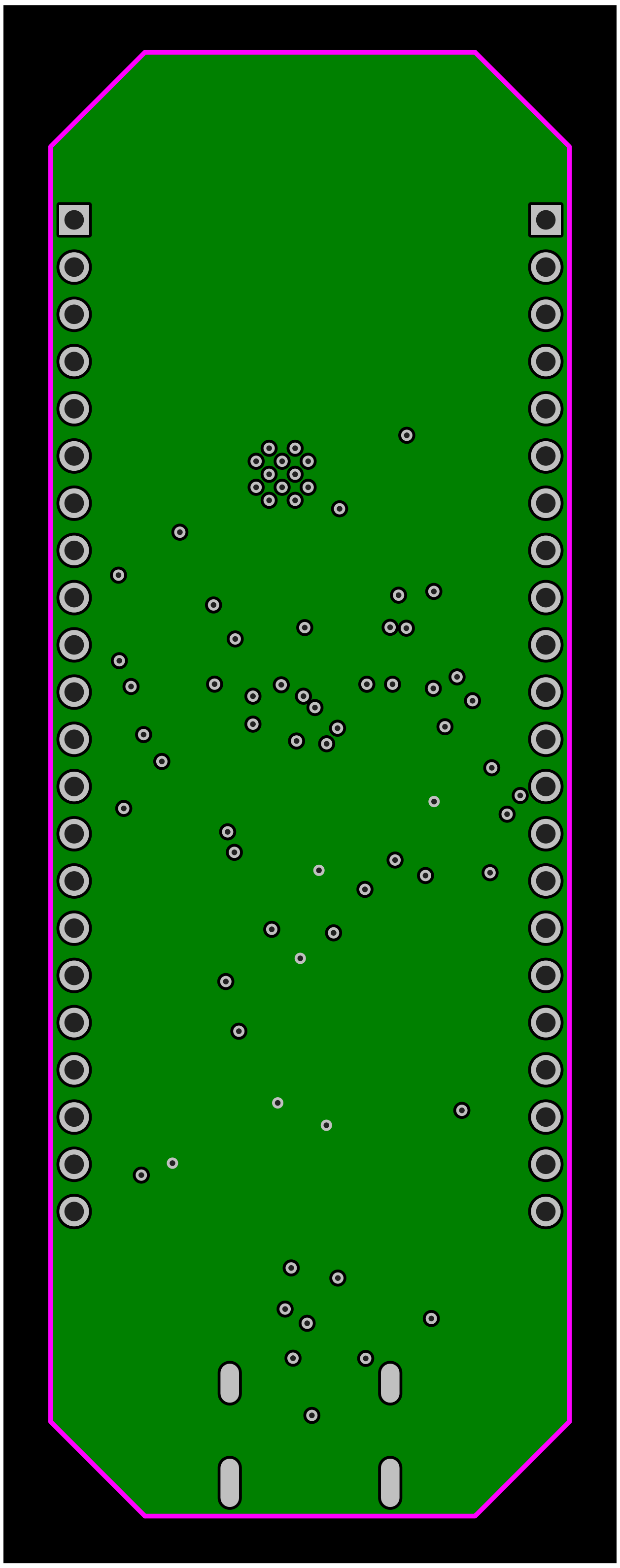


Figure Appendix 2.10: PCB Gerber Layer for the Inner2 - VCC

2.2.9 Multi-Layer

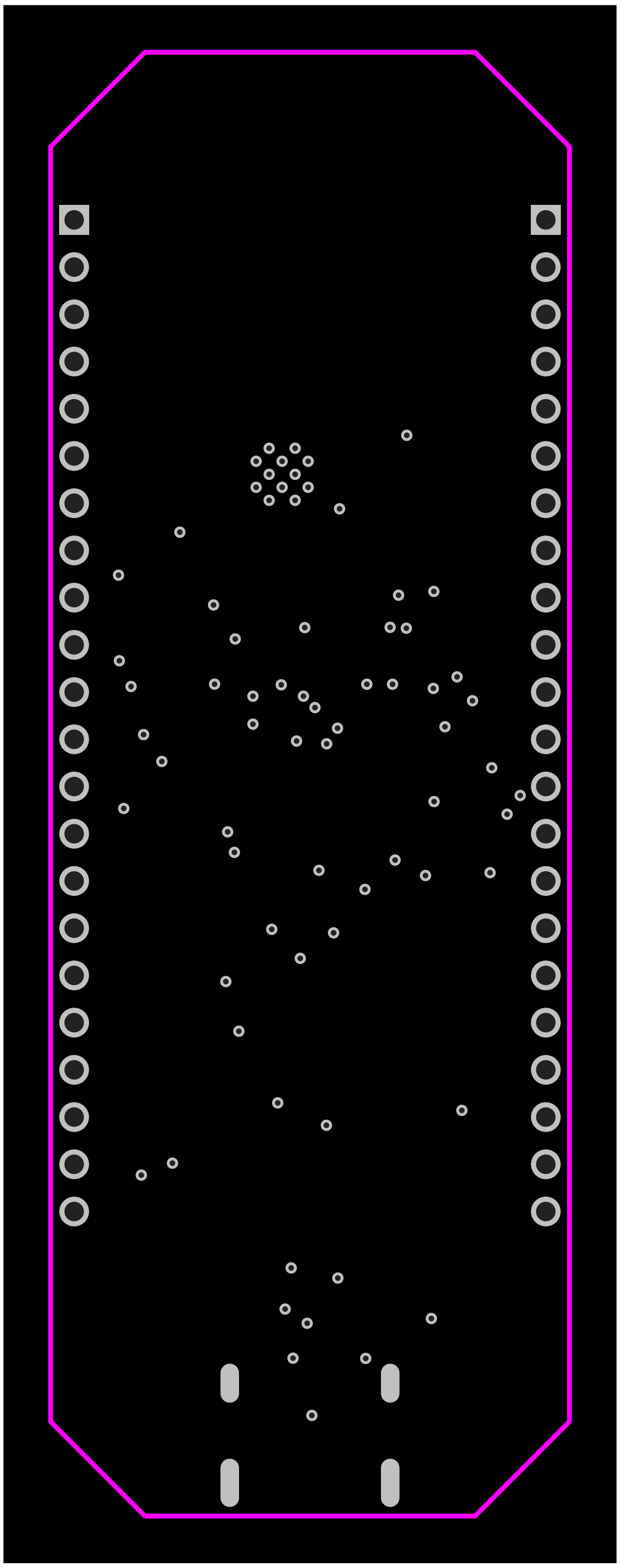


Figure Appendix 2.11: PCB Gerber Layer for the Multi-Layer

2.2.10 Top Layer

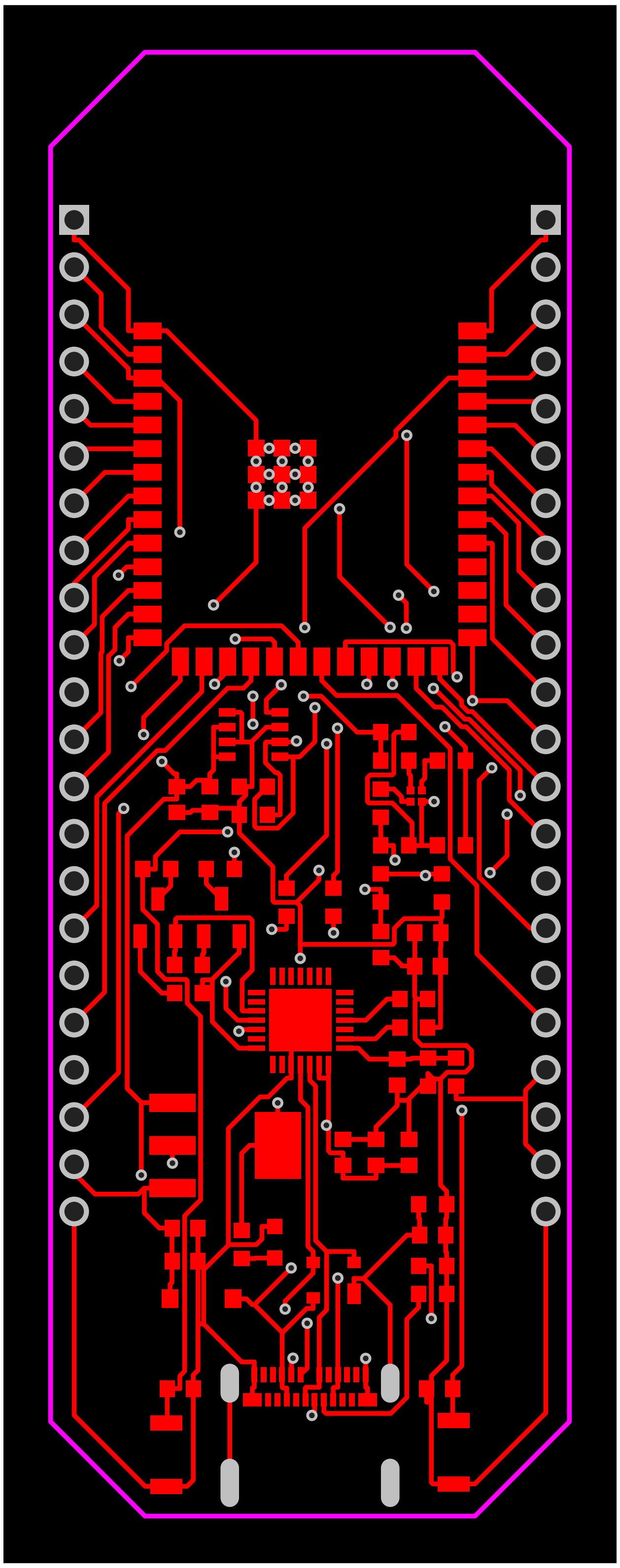


Figure Appendix 2.12: PCB Gerber Layer for the Top Layer

2.2.11 Top Paste Mask Layer

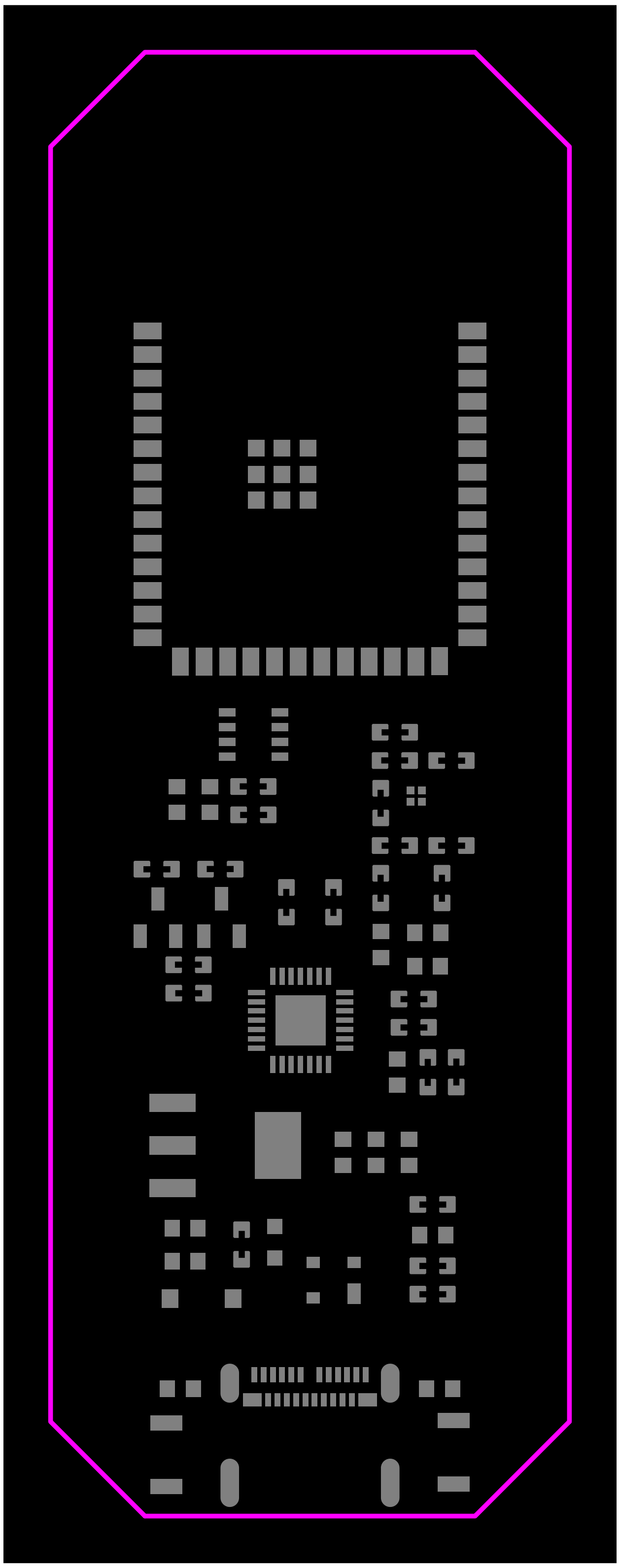


Figure Appendix 2.13: PCB Gerber Layer for the Top Paste Mask Layer

2.2.12 Top Silk Layer

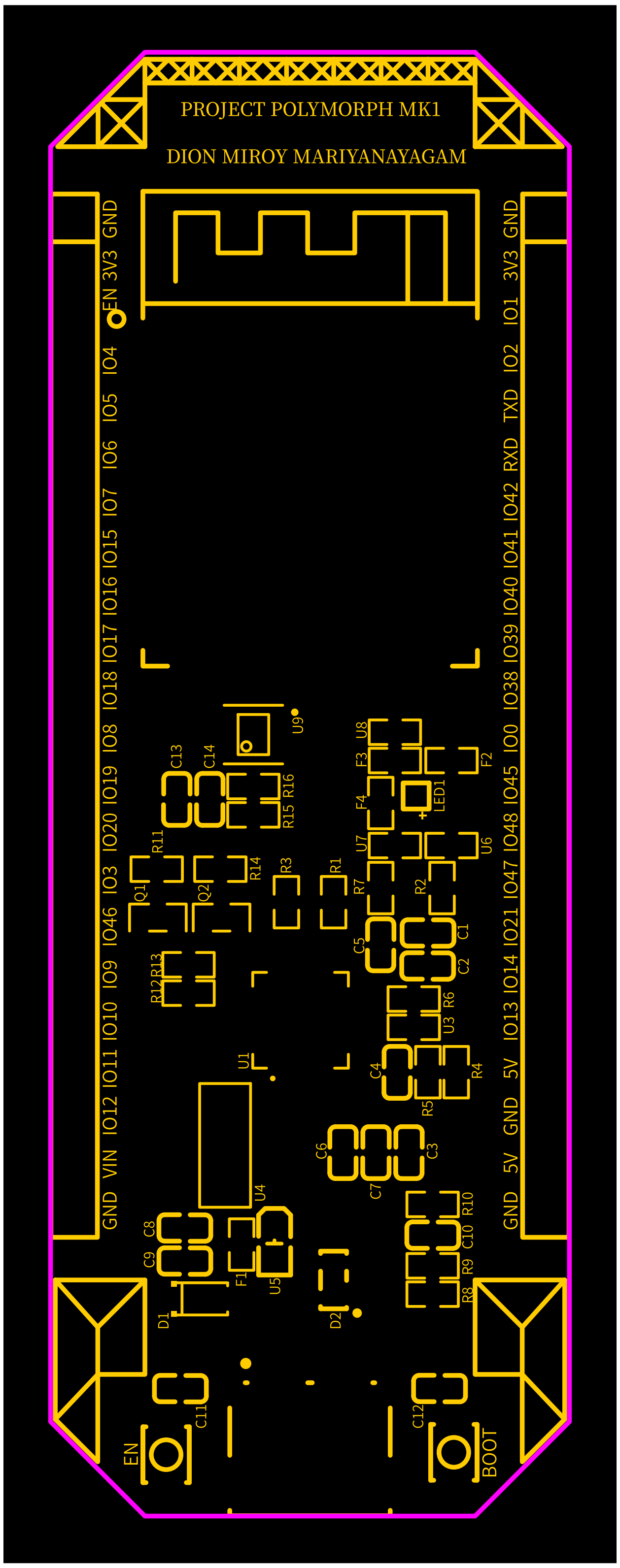

PROJECT POLYMORPH MK1
DION MIROY MARIYANAYAGAM
GND 3V3 EN IO4 IO5 IO6 IO7 IO15 IO16 IO17 IO18 IO8 IO19 IO20 IO3 IO46 IO9 IO10 IO11 IO12 VIN GND
GND 3V3 IO1 IO2 TXD RXD IO42 IO41 IO40 IO39 IO38 IO0 IO45 IO48 IO47 IO21 IO14 IO13 5V GND 5V GND
EN
BOOT

Figure Appendix 2.14: PCB Gerber Layer for the Top Silk Layer

2.2.13 Top Solder Mask Layer

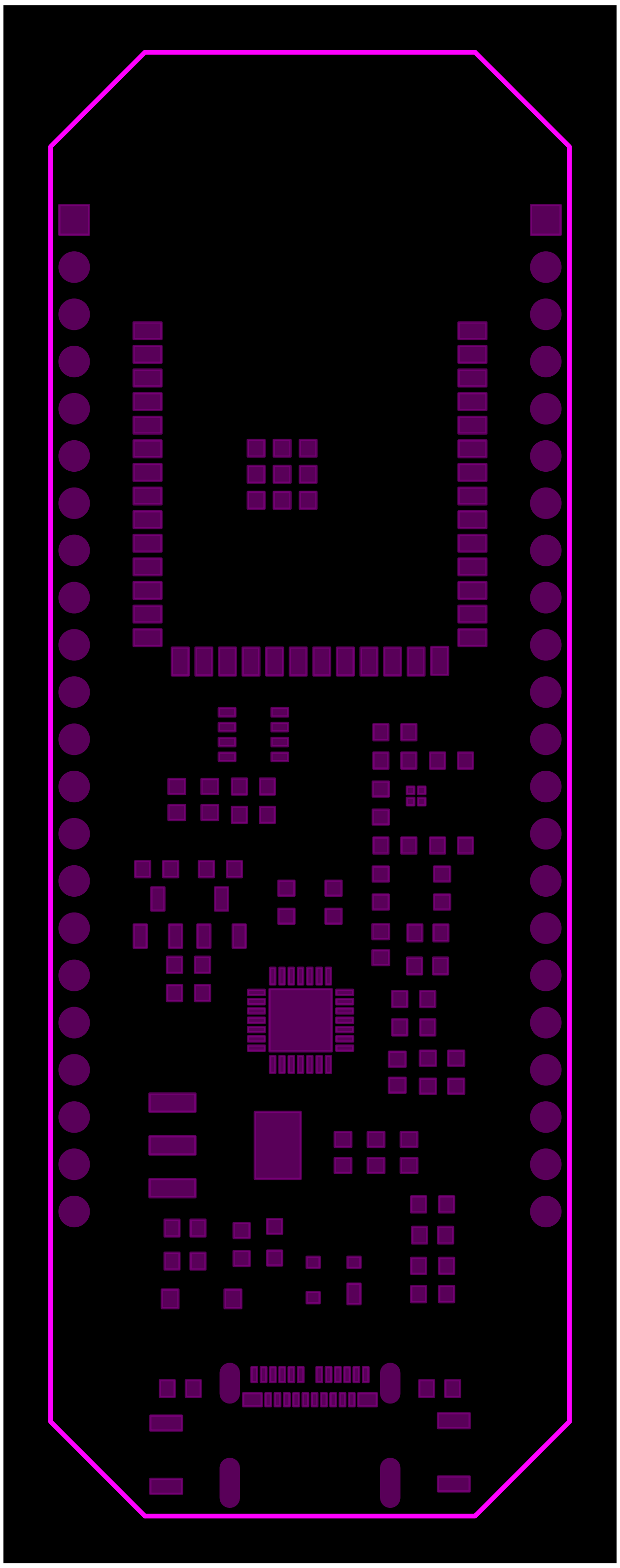


Figure Appendix 2.15: PCB Gerber Layer for the Top Solder Mask Layer

2.2.14 GERBER View

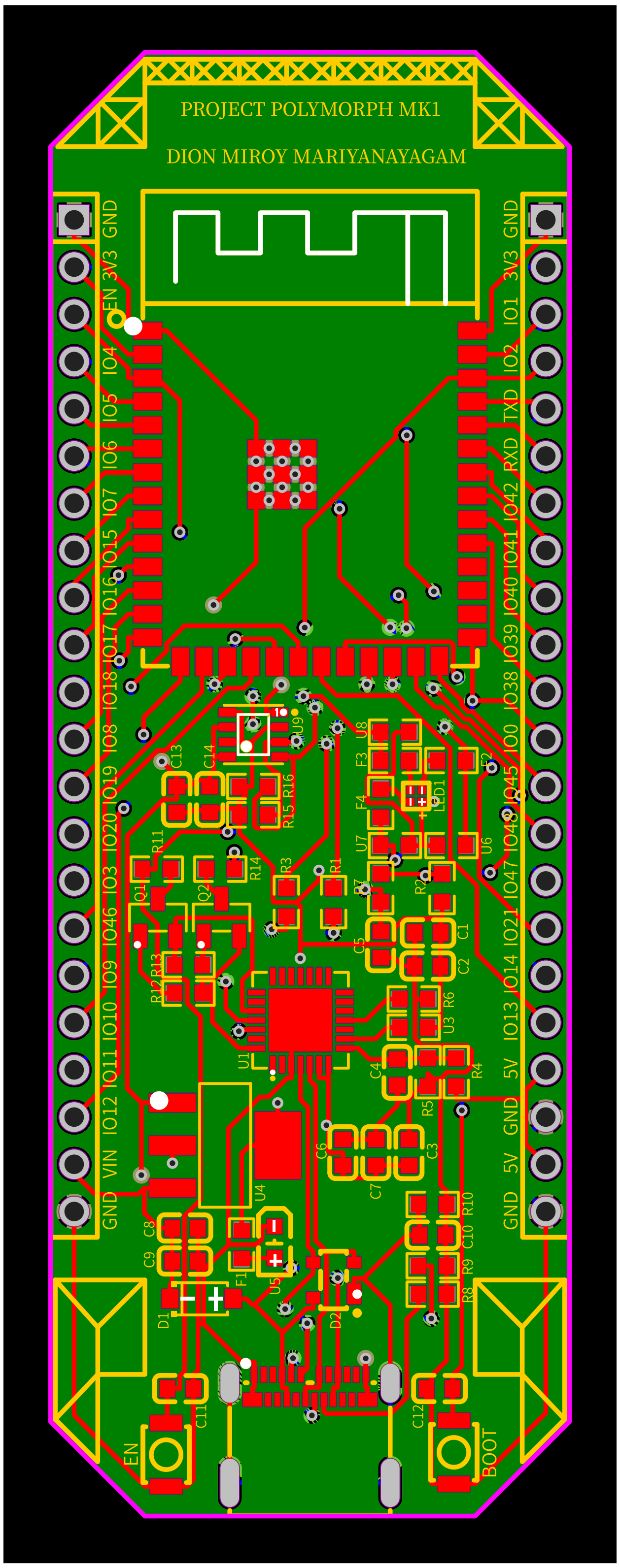


Figure Appendix 2.16: PCB Gerber Layer for the GERBER View

2.2.15 2D Photo View

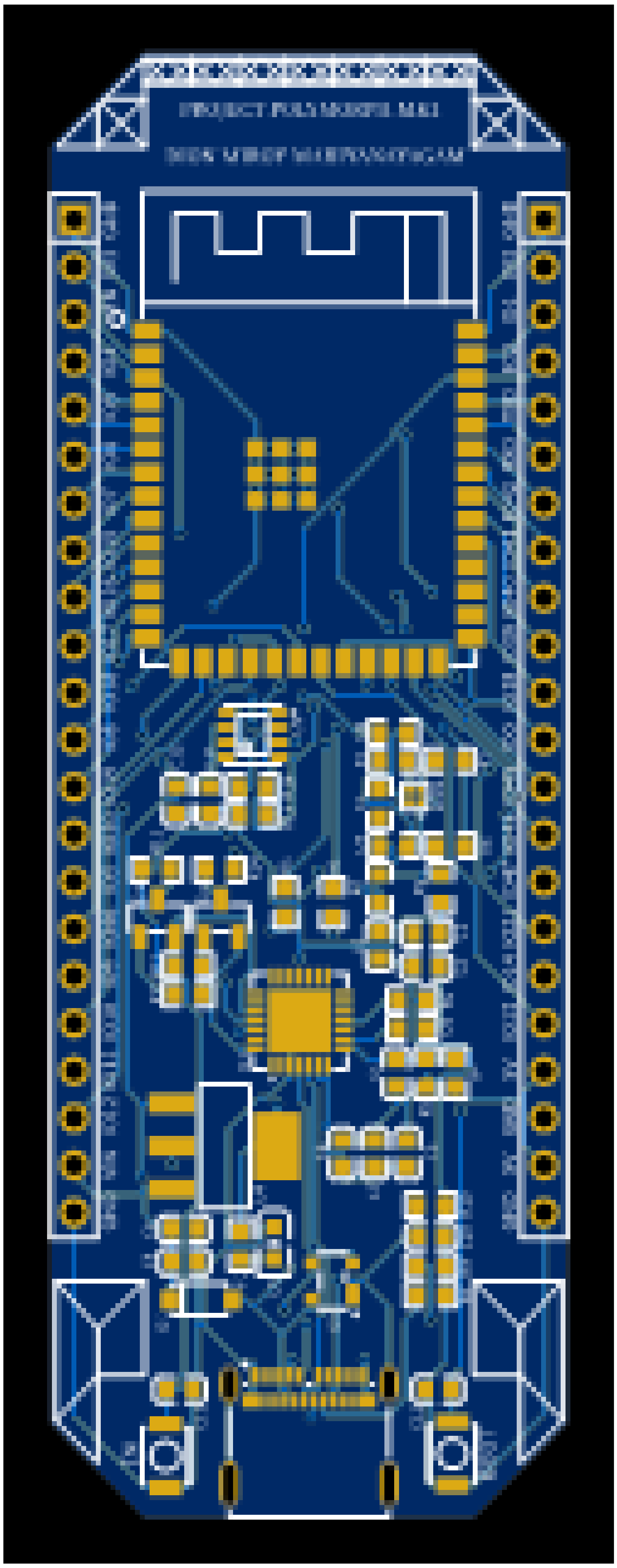


Figure Appendix 2.17: PCB Gerber Layer for the 2D Photo View

2.3 BoM – Bill of Materials

Table 8: Bill of Materials For Custom Development Board

ID	Name	Designator	Footprint	Quantity	Manufacturer Part	Manufacturer	Supplier	Supplier Part	Price
1	PRTR5V0U2X,215	D2	SOT-143_L2.9-W1.3-P1.92-LS2.3-BR	1	PRTR5V0U2X,215	Nexperia(安世)	LCSC	C12333	0.092
2	0.1uF	C2,C7,C9,C11,C12,C13,C14	C0603	7	CL10B104KB8NNNC	SAMSUNG(三星)	LCSC	C1591	0.002
3	10uF	C8,C1,C3,C6	C0603	4	CL10A106MA8NRNC	SAMSUNG(三星)	LCSC	C1591	0.002
4	22.1kΩ	R4	R0603	1	RC0603FR-0747KL	YAGEO(国巨)	LCSC	C105579	0.001
5	1kΩ	U3,U6,U7,U8	R0603	4	AT0603FRE0722KL	YAGEO(国巨)	LCSC	C856613	0.02
6	X6511WV-22H-C30D60	H1,H2	HDR-TH_22P-P2.54-V-M	2	X6511WV-22H-C30D60	XKB Connectivity(中国星坤)	LCSC	C725954	0.335
7	UHD1110-FKA-CL1A13R3Q1BBQFMF3	LED1	LED-SMD_4P-L1.0-W1.0_UHD1110-FKA-CL1A13R3Q1BBQFMF3	1	UHD1110-FKA-CL1A13r3q1BBQFMF3	null	LCSC	C9900019213	
8	BCW65CLT1G	Q1,Q2	SOT-23-3_L2.9-W1.3-P1.90-LS2.4-BR	2	BCW65CLT1G	onsemi(安森美)	LCSC	C232540	0.078
9	105450-0101	USBC1	USB-C-SMD_TYPE-C-USB-18	1	1054500101	MOLEX	LCSC	C134092	0.694
10	1uF	C4,C5	C0603	2	CL10A105KO8NNNC	SAMSUNG(三星)	LCSC	C1592	0.003
11	4.7nF	C10	C0603	1	CL10B104KB8NNNC	SAMSUNG(三星)	LCSC	C1591	0.002
12	PMEG6020AELRX	D1	SOD-123_L2.6-W1.7-LS3.5-RD	1	PMEG6020AELRX	Nexperia(安世)	LCSC	C552938	0.212
13	500Ω	F1,F2,F3,F4	R0603	4	RT0603BRC07500RL	YAGEO(国巨)	LCSC	C860931	0.088
14	0Ω	R1,R2,R3,R11,R14	R0603	5	RC0603FR-070RL	YAGEO(国巨)	LCSC	C100044	
15	47.5kΩ	R5	R0603	1	RC0603FR-0747KL	YAGEO(国巨)	LCSC	C105579	0.001
16	10kΩ	R6,R7,R12,R13	R0603	4	RC0603FR-0710KL	YAGEO(国巨)	LCSC	C98220	0.001
17	5.1kΩ	R8,R9	R0603	2	RT0603BRD075K1L	YAGEO(国巨)	LCSC	C122969	0.025
18	1MΩ	R10	R0603	1	RT0603BRD071ML	YAGEO(国巨)	LCSC	C326730	0.023
19	4.7kΩ	R15,R16	R0603	2	RT0603BRD075K1L	YAGEO(国巨)	LCSC	C122969	0.025
20	B3U-1000P	EN,BOOT	KEY-SMD_B3U-1000PM	2	B3U-1000P	OMRON(欧姆龙)	LCSC	C231329	0.155
21	CP2102N-A02-GQFN28	U1	QFN-28_L5.0-W5.0-P0.50-TL-EP3.3	1	CP2102N-A02-GQFN28	SILICON LABS(芯科)	LCSC	C1550553	2.452
22	ESP32-S3-WROOM-2-N32R8V	U2	WIRELM-SMD_ESP32-S3-WROOM-1	1	ESP32-S3-WROOM-2-N32R8V	ESPRESSIF(乐鑫)	LCSC	C3021270	
23	LDL1117S33R	U4	SOT-223-4_L6.5-W3.5-P2.30-LS7.0-BR	1	LDL1117S33R	ST(意法半导体)	LCSC	C435835	0.174
24	SMLD12BN1WT86	U5	LED0603-R-RD_BLUE	1	SMLD12BN1WT86	ROHM(罗姆)	LCSC	C2837822	0.21
25	BME688	U9	LGA-8_L3.0-W3.0-P0.80-TR-1	1	BME688	null	LCSC	C9900016822	

2.4 Implementation Process

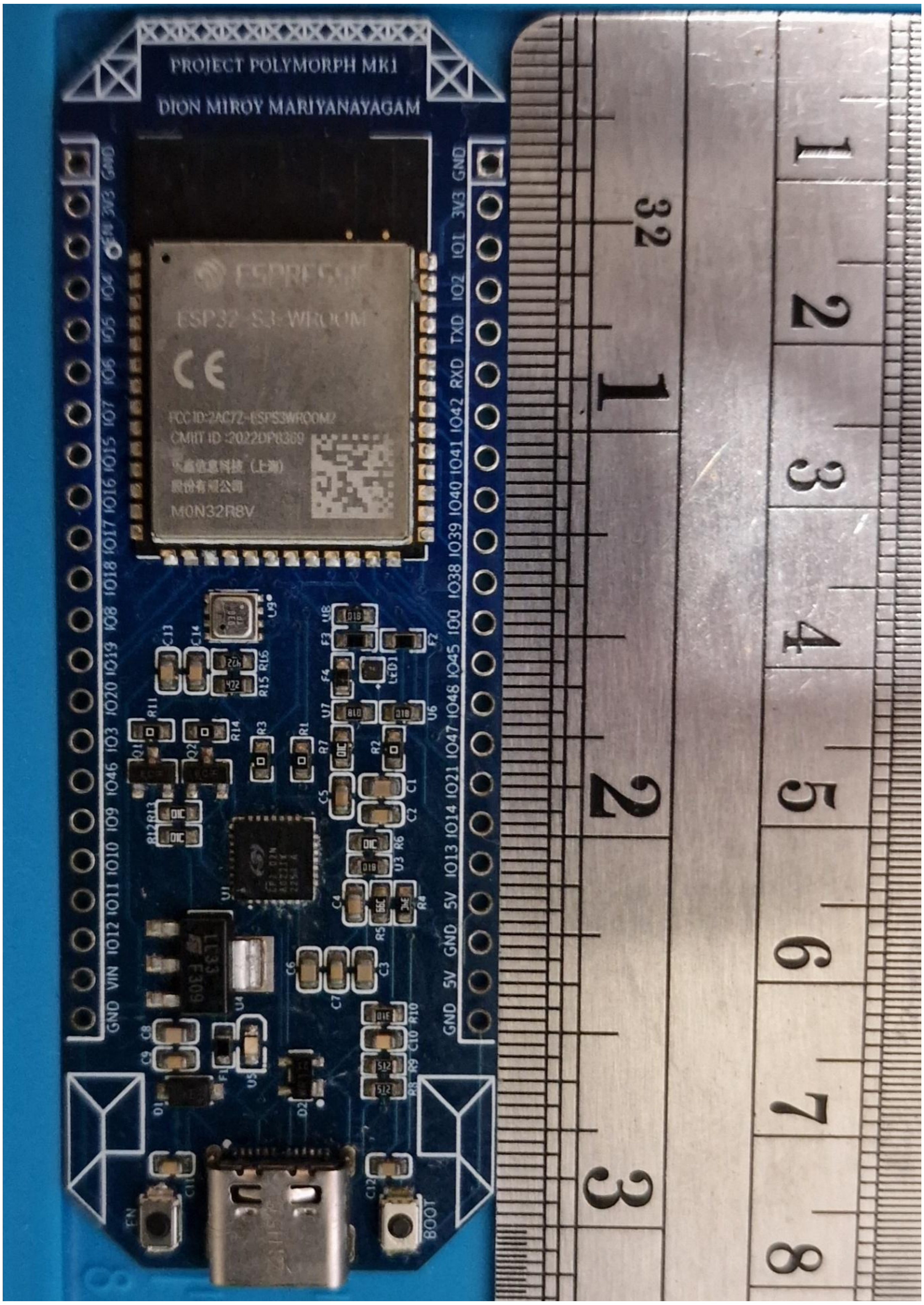

PROJECT POLYMORPH MK1
DION MIROY MARIYANAYAGAM
ESP32-S3-WROOM
GND 3V3 IO1 IO2 TXD RXD IO42 IO41 IO40 IO39 IO38 IO0 IO45 IO48 IO47 IO21 IO14 IO13 5V GND 5V GND
GND 3V3 EN IO4 IO5 IO6 IO7 IO15 IO16 IO17 IO18 IO8 IO19 IO20 IO3 IO46 IO9 IO10 IO11 IO12 VIN GND
EN
BOOT

Figure Appendix 2.18: Implementation Process

Appendix 3: Basic Server and Client Communication

3.1 Server-side code

```
#include <WiFi.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server settings
WiFiServer server(80); // Port 80 is the default HTTP port

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Start the server
  server.begin();
  Serial.println("Server started.");
}

void loop()
{
  // Check if a client has connected
  WiFiClient client = server.available();

  if (client)
  {
    Serial.println("New Client connected.");
    String currentLine = "";

    while (client.connected())
    {
      if (client.available())
      {
        char c = client.read();
        Serial.write(c); // Display the data received from the client

        if (c == '\n')
        {
          if (currentLine.length() == 0)
          {
            // Respond to the client
            client.println("HTTP/1.1 200 OK");
            client.println("Content-type:text/html");
            client.println();
            client.println("<!DOCTYPE HTML>");
            client.println("<html><body><h1>Hello from ESP32
Server!</h1></body></html>");
            break;
          }
          else
          {
            currentLine = "";
          }
        }
        else if (c != '\r')
        {
          currentLine += c;
        }
      }
    }

    // Give the client time to receive the data
    delay(10);
    client.stop();
    Serial.println("Client disconnected.");
  }
}
```

3.2 Client-side code

```
#include <WiFi.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server IP and Port (Replace with the server IP address and port)
const char *host = "192.168.1.10"; // Replace with your server's IP address
const uint16_t port = 80;

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Connect to the server
  Serial.print("Connecting to ");
  Serial.print(host);
  Serial.print(":");
  Serial.println(port);

  WiFiClient client;

  if (client.connect(host, port))
  {
    Serial.println("Connected to server.");

    // Send a request to the server
    client.println("The quick brown fox jumps over the lazy dog
1234567890!@#$%^&*()_+-=[]{}|;':,.<>/?\n\t");

    // Wait for the server response
    while (client.connected() && !client.available())
    {
      delay(10);
    }

    while (client.available())
    {
      String line = client.readStringUntil('\r');
      Serial.print(line);
    }

    Serial.println();
    Serial.println("Disconnecting from server...");
    client.stop();
  }
  else
  {
    Serial.println("Connection to server failed.");
  }
}

void loop()
{
  // Add code here to periodically connect and communicate with the server
  delay(10000); // Wait 10 seconds between connections
}
```

Appendix 4: Adaptive Amoeba Complexity

```
#include <Arduino.h>
#include <stdint.h>

// Definitions for various encryption configurations
#define SPECK_ROUNDS_128 22
#define SPECK_ROUNDS_256 32
#define SIMON_ROUNDS_128 32
#define SIMON_ROUNDS_256 44

// Define key sizes in 32-bit words for SPECK and SIMON
#define SPECK_KEY_WORDS 4 // SPECK uses 4 words (128 bits key)
#define SIMON_KEY_WORDS 4 // SIMON uses 4 words (128 bits key)

char data[] = "The quick brown fox jumps over the lazy dog
1234567890!@#$%^&*()_+-=[]{}|;':,.<>/?\n\t";
unsigned long start_time, end_time;

extern unsigned int __heap_start;
extern void *__brkval;

int freeMemory()
{
  int v;
  return (int)&v - (__brkval == 0 ? (int)&__heap_start : (int)__brkval);
}

// XOR Encryption
void xor_encryption()
{
  char key = 0xAA;
  char data_copy[sizeof(data)];
  strcpy(data_copy, data);

  start_time = micros();
  for (int i = 0; i < sizeof(data_copy) - 1; i++)
  {
    data_copy[i] ^= key;
  }
  end_time = micros();

  Serial.print("XOR Encryption: ");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
}

// Caesar Cipher
void caesar_cipher()
```

```
{
  int shift = 3;
  char data_copy[sizeof(data)];
  strcpy(data_copy, data);

  start_time = micros();
  for (int i = 0; i < sizeof(data_copy) - 1; i++)
  {
    if (isalpha(data_copy[i]))
    {
      char offset = isupper(data_copy[i]) ? 'A' : 'a';
      data_copy[i] = (data_copy[i] - offset + shift) % 26 + offset;
    }
  }
  end_time = micros();

  Serial.print("Caesar Cipher: ");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
}

// ROT13 Encryption
void rot13_encryption()
{
  char data_copy[sizeof(data)];
  strcpy(data_copy, data);

  start_time = micros();
  for (int i = 0; i < sizeof(data_copy) - 1; i++)
  {
    if (isalpha(data_copy[i]))
    {
      char offset = isupper(data_copy[i]) ? 'A' : 'a';
      data_copy[i] = (data_copy[i] - offset + 13) % 26 + offset;
    }
  }
  end_time = micros();

  Serial.print("ROT13 Encryption: ");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
}

// SPECK Encryption
void speck_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speck_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speck_round(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speck_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    speck_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

void speck_encryption(uint8_t key_size)
{
  uint8_t rounds = (key_size == 128) ? SPECK_ROUNDS_128 : SPECK_ROUNDS_256;
  uint32_t key[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
  uint32_t key_schedule[rounds];
  uint32_t plaintext[2], ciphertext[2];

  speck_expand(key, key_schedule, rounds);

  start_time = micros();
  for (int i = 0; i < sizeof(data) - 1; i += 8)
  {
    memcpy(plaintext, data + i, 8);
    speck_encrypt(plaintext, ciphertext, key_schedule, rounds);
  }
  end_time = micros();

  Serial.print("SPECK Encryption (");
  Serial.print(key_size);
  Serial.print(" bits): ");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
}

// SIMON Encryption
void simon_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}

void simon_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simon_round(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simon_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    simon_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}
```

```
void simon_encryption(uint8_t key_size)
{
  uint8_t rounds = (key_size == 128) ? SIMON_ROUNDS_128 : SIMON_ROUNDS_256;
  uint32_t key[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
  uint32_t key_schedule[rounds];
  uint32_t plaintext[2], ciphertext[2];

  simon_expand(key, key_schedule, rounds);

  start_time = micros();
  for (int i = 0; i < sizeof(data) - 1; i += 8)
  {
    memcpy(plaintext, data + i, 8);
    simon_encrypt(plaintext, ciphertext, key_schedule, rounds);
  }
  end_time = micros();

  Serial.print("SIMON Encryption (");
  Serial.print(key_size);
  Serial.print(" bits): ");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
}

void setup()
{
  Serial.begin(115200);

  int available_memory = freeMemory();
  Serial.print("Available memory: ");
  Serial.println(available_memory);

#if defined(ARDUINO_ARCH_ESP32)
  Serial.println("ESP32 detected");
  if (available_memory > 10000)
  {
    speck_encryption(256);
    simon_encryption(256);
  }
  else
  {
    speck_encryption(128);
    simon_encryption(128);
  }
#elif defined(ARDUINO_ARCH_AVR)
  Serial.println("Arduino Nano (AVR) detected");
  xor_encryption();
  caesar_cipher();
  rot13_encryption();
#elif defined(ARDUINO_ARCH_STM32)
  Serial.println("STM32 detected");
  if (available_memory > 5000)
  {
    speck_encryption(256);
    simon_encryption(256);
  }
  else
  {
    speck_encryption(128);
    simon_encryption(128);
  }
#elif defined(ARDUINO_ARCH_SAMD)
  Serial.println("SAMD detected (Arduino Zero, MKR series)");
  if (available_memory > 6000)
  {
    speck_encryption(256);
    simon_encryption(256);
  }
  else
  {
    speck_encryption(128);
    simon_encryption(128);
  }
#elif defined(TEENSYDUINO)
  Serial.println("Teensy detected");
  if (available_memory > 10000)
  {
    speck_encryption(256);
    simon_encryption(256);
  }
  else
  {
    speck_encryption(128);
    simon_encryption(128);
  }
#elif defined(ARDUINO_ARCH_ESP8266)
  Serial.println("ESP8266 detected");
  if (available_memory > 5000)
  {
    speck_encryption(256);
    simon_encryption(256);
  }
  else
  {
    speck_encryption(128);
    simon_encryption(128);
  }
#else
  Serial.println("Unknown board detected");
  // Default to simple encryption method
s
  xor_encryption();
#endif
}

void loop()
{
  // Not needed for this test
}
```

Appendix 5: Encrypted Server and Client Communication

5.1 Server-side code – Lightweight Encryption

```
#include <WiFi.h>
#include <stdint.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server settings
WiFiServer server(80); // Port 80 is the default HTTP port

// Definitions for various encryption configurations
#define SPECK_ROUNDS_128 22
#define SPECK_ROUNDS_256 32
#define SIMON_ROUNDS_128 32
#define SIMON_ROUNDS_256 44

// Define key sizes in 32-bit words for SPECK and SIMON
#define SPECK_KEY_WORDS 4 // SPECK uses 4 words (128 bits key)
#define SIMON_KEY_WORDS 4 // SIMON uses 4 words (128 bits key)

// XOR Encryption/Decryption
void xor_crypt(char *data, char key)
{
  for (int i = 0; i < strlen(data); i++)
  {
    data[i] ^= key;
  }
}

// Caesar Cipher Encryption/Decryption
void caesar_crypt(char *data, int shift)
{
  for (int i = 0; i < strlen(data); i++)
  {
    if (isalpha(data[i]))
    {
      char offset = isupper(data[i]) ? 'A' : 'a';
      data[i] = (data[i] - offset + shift) % 26 + offset;
    }
  }
}

// ROT13 Encryption/Decryption
void rot13_crypt(char *data)
{
  caesar_crypt(data, 13);
```

```
}

// SPECK Encryption
void speck_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speck_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speck_round(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speck_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    speck_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

// SIMON Encryption
void simon_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
```

```
}

void simon_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simon_round(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simon_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    simon_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Start the server
  server.begin();
  Serial.println("Server started.");
}

void loop()
{
  // Check if a client has connected
  WiFiClient client = server.available();

  if (client)
  {
    Serial.println("New Client connected.");
    char data[] = "Hello from ESP32 Server!";
    char key = 0xAA; // Key for XOR
    int shift = 3;   // Shift for Caesar Cipher

    // Example: Encrypt the message using XOR
    xor_crypt(data, key);

    // Send the encrypted message to the client
    client.write((const uint8_t *)data, strlen(data));
    Serial.print("Encrypted message sent: ");
    Serial.println(data);

    // Disconnect the client
    client.stop();
    Serial.println("Client disconnected.");
  }
}
```

5.2 Client-side code – Lightweight Encryption

```
#include <WiFi.h>
#include <stdint.h>

// Replace with your network credentials
const char *ssid = "Your_SSID";
const char *password = "Your_PASSWORD";

// Server IP and Port (Replace with the server IP address and port)
const char *host = "192.168.1.10"; // Replace with your server's IP address
const uint16_t port = 80;

// XOR Decryption
void xor_crypt(char *data, char key)
{
  for (int i = 0; i < strlen(data); i++)
  {
    data[i] ^= key;
  }
}

// Caesar Cipher Decryption
void caesar_crypt(char *data, int shift)
{
  caesar_crypt(data, 26 - shift);
}

// ROT13 Decryption
void rot13_crypt(char *data)
{
  caesar_crypt(data, 13);
}

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Connect to the server
  Serial.print("Connecting to ");
  Serial.print(host);
  Serial.print(":");
  Serial.println(port);

  WiFiClient client;

  if (client.connect(host, port))
  {
    Serial.println("Connected to server.");

    // Receive the encrypted message from the server
    char encrypted_data[64];
    int len = client.readBytes(encrypted_data, sizeof(encrypted_data) - 1);
    encrypted_data[len] = '\0';

    Serial.print("Encrypted message received: ");
    Serial.println(encrypted_data);

    // Example: Decrypt the message using XOR
    char key = 0xAA; // Key for XOR
    xor_crypt(encrypted_data, key);

    Serial.print("Decrypted message: ");
    Serial.println(encrypted_data);

    Serial.println("Disconnecting from server...");
    client.stop();
  }
  else
  {
    Serial.println("Connection to server failed.");
  }
}

void loop()
{
  // Add code here to periodically connect and communicate with the server
  delay(10000); // Wait 10 seconds between connections
}
```

5.3 Server-side code – Block Cipher Encryption

```
#include <WiFi.h>
#include <stdint.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server settings
WiFiServer server(80); // Port 80 is the default HTTP port

// Definitions for various encryption configurations
#define SPECK_ROUNDS_128 22
#define SPECK_ROUNDS_256 32
#define SIMON_ROUNDS_128 32
#define SIMON_ROUNDS_256 44

// Define key sizes in 32-bit words for SPECK and SIMON
#define SPECK_KEY_WORDS 4 // SPECK uses 4 words (128 bits key)
#define SIMON_KEY_WORDS 4 // SIMON uses 4 words (128 bits key)

// SPECK Encryption Functions
void speck_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speck_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speck_round(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}
```

```
void speck_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    speck_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

// SIMON Encryption Functions
void simon_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}

void simon_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simon_round(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simon_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    simon_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
```

```
}

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Start the server
  server.begin();
  Serial.println("Server started.");
}

void loop()
{
  // Check if a client has connected
  WiFiClient client = server.available();

  if (client)
  {
    Serial.println("New Client connected.");

    // Choose encryption method
    bool use_speck = true; // Set to false to use SIMON

    char message[] = "Hello from ESP32 Server!";
    uint32_t plaintext[2], ciphertext[2];
    memcpy(plaintext, message, 8); // Encrypt the first 8 bytes for simplicity

    if (use_speck)
    {
      // SPECK Encryption
      uint8_t rounds = SPECK_ROUNDS_128;
      uint32_t key[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
      uint32_t key_schedule[rounds];

      speck_expand(key, key_schedule, rounds);
      speck_encrypt(plaintext, ciphertext, key_schedule, rounds);

      client.write((const uint8_t *)ciphertext, sizeof(ciphertext));
      Serial.println("Encrypted message sent with SPECK.");
    }
    else
    {
      // SIMON Encryption
      uint8_t rounds = SIMON_ROUNDS_128;
      uint32_t key[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
      uint32_t key_schedule[rounds];

      simon_expand(key, key_schedule, rounds);
      simon_encrypt(plaintext, ciphertext, key_schedule, rounds);

      client.write((const uint8_t *)ciphertext, sizeof(ciphertext));
      Serial.println("Encrypted message sent with SIMON.");
    }

    // Disconnect the client
    client.stop();
    Serial.println("Client disconnected.");
  }
}
```

5.4 Client-side code – Block Cipher Encryption

```
#include <WiFi.h>
#include <stdint.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server IP and Port (Replace with the server IP address and port)
const char *host = "192.168.1.10"; // Replace with your server's IP address
const uint16_t port = 80;

// Definitions for various encryption configurations
#define SPECK_ROUNDS_128 27 // Correct rounds for SPECK-128/128
#define SIMON_ROUNDS_128 44 // Correct rounds for SIMON-128/128
#define SPECK_KEY_WORDS 4   // SPECK uses 4 words (128 bits key)
#define SIMON_KEY_WORDS 4   // SIMON uses 4 words (128 bits key)

// SPECK Key Expansion
void speck_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    uint32_t &x = a[i % (SPECK_KEY_WORDS - 1)];
    b = (b >> 8) | (b << (32 - 8));
    b += x;
    b ^= i;
    x = (x << 3) | (x >> (32 - 3));
    x ^= b;
    S[i + 1] = b;
  }
}

// SPECK Decryption Functions
void speck_round_inv(uint32_t &x, uint32_t &y, uint32_t k)
{
  y ^= x;
  y = (y >> 3) | (y << (32 - 3));
  x ^= k;
  x -= y;
  x = (x << 8) | (x >> (32 - 8));
}

void speck_decrypt(uint32_t ciphertext[2], uint32_t plaintext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  plaintext[0] = ciphertext[0];
  plaintext[1] = ciphertext[1];
  for (i = rounds; i > 0; i--)
  {
    speck_round_inv(plaintext[1], plaintext[0], key_schedule[i - 1]);
  }
}

// SIMON Key Expansion
void simon_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    uint32_t &x = a[i % (SIMON_KEY_WORDS - 1)];
    uint32_t tmp = (b << 1) | (b >> (32 - 1));
    tmp &= (tmp << 8);
    tmp ^= x;
    x ^= i;
    x = (x >> 1) | (x << (32 - 1));
    S[i + 1] = b;
    b = tmp;
  }
}

// SIMON Decryption Functions
void simon_round_inv(uint32_t &x, uint32_t &y, uint32_t k)
{
  y ^= x;
  x ^= k;
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  y = tmp ^ y;
  x = (x >> 1) | (x << (32 - 1));
}
```

```
void simon_decrypt(uint32_t ciphertext[2], uint32_t plaintext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  plaintext[0] = ciphertext[0];
  plaintext[1] = ciphertext[1];
  for (i = rounds; i > 0; i--)
  {
    simon_round_inv(plaintext[1], plaintext[0], key_schedule[i - 1]);
  }
}

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Connect to the server
  Serial.print("Connecting to ");
  Serial.print(host);
  Serial.print(":");
  Serial.println(port);

  WiFiClient client;

  if (client.connect(host, port))
  {
    Serial.println("Connected to server.");

    uint32_t ciphertext[2];
    int len = client.readBytes((char *)ciphertext, sizeof(ciphertext));
    if (len != sizeof(ciphertext))
    {
      Serial.println("Error: Did not receive expected ciphertext size.");
      return;
    }

    // Choose decryption method
    bool use_speck = true; // Set to false to use SIMON

    uint32_t plaintext[2];

    if (use_speck)
    {
      // SPECK Decryption
      uint8_t rounds = SPECK_ROUNDS_128;
      uint32_t key[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
      uint32_t key_schedule[rounds];

      speck_expand(key, key_schedule, rounds);
      speck_decrypt(ciphertext, plaintext, key_schedule, rounds);

      Serial.println("Decrypted message with SPECK:");
      Serial.println((char *)plaintext);
    }
    else
    {
      // SIMON Decryption
      uint8_t rounds = SIMON_ROUNDS_128;
      uint32_t key[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
      uint32_t key_schedule[rounds];

      simon_expand(key, key_schedule, rounds);
      simon_decrypt(ciphertext, plaintext, key_schedule, rounds);

      Serial.println("Decrypted message with SIMON:");
      Serial.println((char *)plaintext);
    }

    Serial.println("Disconnecting from server...");
    client.stop();
  }
  else
  {
    Serial.println("Connection to server failed.");
  }
}

void loop()
```

```
{
  // Add code here to periodically connect and communicate with the server
  delay(10000); // Wait 10 seconds between connections
}
```

Appendix 6: Sensor Encrypted Communication For Server and Client

6.1 Lightweight server-side code

```
#include <WiFi.h>
#include <stdint.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server settings
WiFiServer server(80); // Port 80 is the default HTTP port

// Definitions for various encryption configurations
#define SPECK_ROUNDS_128 22
#define SPECK_ROUNDS_256 32
#define SIMON_ROUNDS_128 32
#define SIMON_ROUNDS_256 44

// Define key sizes in 32-bit words for SPECK and SIMON
#define SPECK_KEY_WORDS 4 // SPECK uses 4 words (128 bits key)
#define SIMON_KEY_WORDS 4 // SIMON uses 4 words (128 bits key)

// XOR Encryption/Decryption
void xor_crypt(char *data, char key)
{
  for (int i = 0; i < strlen(data); i++)
  {
    data[i] ^= key;
  }
}

// Caesar Cipher Encryption/Decryption
void caesar_crypt(char *data, int shift)
{
  for (int i = 0; i < strlen(data); i++)
  {
    if (isalpha(data[i]))
    {
      char offset = isupper(data[i]) ? 'A' : 'a';
      data[i] = (data[i] - offset + shift) % 26 + offset;
    }
  }
}

// ROT13 Encryption/Decryption
void rot13_crypt(char *data)
{
  caesar_crypt(data, 13);
```

```
}

// SPECK Encryption
void speck_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speck_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speck_round(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speck_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    speck_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

// SIMON Encryption
void simon_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
```

```
}

void simon_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simon_round(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simon_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    simon_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Start the server
  server.begin();
  Serial.println("Server started.");
}

void loop()
{
  // Check if a client has connected
  WiFiClient client = server.available();

  if (client)
  {
    Serial.println("New Client connected.");
    char data[] = "Hello from ESP32 Server!";
    char key = 0xAA; // Key for XOR
    int shift = 3;   // Shift for Caesar Cipher

    // Example: Encrypt the message using XOR
    xor_crypt(data, key);

    // Send the encrypted message to the client
    client.write((const uint8_t *)data, strlen(data));
    Serial.print("Encrypted message sent: ");
    Serial.println(data);

    // Disconnect the client
    client.stop();
    Serial.println("Client disconnected.");
  }
}
```

6.2 Lightweight client-side code

```
#include <WiFi.h>
#include <stdint.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server settings
WiFiServer server(80); // Port 80 is the default HTTP port

// Definitions for various encryption configurations
#define SPECK_ROUNDS_128 22
#define SPECK_ROUNDS_256 32
#define SIMON_ROUNDS_128 32
#define SIMON_ROUNDS_256 44

// Define key sizes in 32-bit words for SPECK and SIMON
#define SPECK_KEY_WORDS 4 // SPECK uses 4 words (128 bits key)
#define SIMON_KEY_WORDS 4 // SIMON uses 4 words (128 bits key)

// XOR Encryption/Decryption
void xor_crypt(char *data, char key)
{
  for (int i = 0; i < strlen(data); i++)
  {
    data[i] ^= key;
  }
}

// Caesar Cipher Encryption/Decryption
void caesar_crypt(char *data, int shift)
{
  for (int i = 0; i < strlen(data); i++)
  {
    if (isalpha(data[i]))
    {
      char offset = isupper(data[i]) ? 'A' : 'a';
      data[i] = (data[i] - offset + shift) % 26 + offset;
    }
  }
}

// ROT13 Encryption/Decryption
void rot13_crypt(char *data)
{
  caesar_crypt(data, 13);
}
```

```
// SPECK Encryption
void speck_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speck_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speck_round(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speck_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    speck_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

// SIMON Encryption
void simon_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}
```

```
void simon_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simon_round(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simon_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    simon_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  // Start the server
  server.begin();
  Serial.println("Server started.");
}

void loop()
{
  // Check if a client has connected
  WiFiClient client = server.available();

  if (client)
  {
    Serial.println("New Client connected.");
    char data[] = "Hello from ESP32 Server!";
    char key = 0xAA; // Key for XOR
    int shift = 3;   // Shift for Caesar Cipher

    // Example: Encrypt the message using XOR
    xor_crypt(data, key);

    // Send the encrypted message to the client
    client.write((const uint8_t *)data, strlen(data));
    Serial.print("Encrypted message sent: ");
    Serial.println(data);

    // Disconnect the client
    client.stop();
    Serial.println("Client disconnected.");
  }
}
```

6.3 Block cipher server-side code

```
#include <WiFi.h>
#include <stdint.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server settings
WiFiServer server(80); // Port 80 is the default HTTP port

// Definitions for various encryption configurations
#define SPECK_ROUNDS_128 22
#define SPECK_ROUNDS_256 32
#define SIMON_ROUNDS_128 32
#define SIMON_ROUNDS_256 44

// Define key sizes in 32-bit words for SPECK and SIMON
#define SPECK_KEY_WORDS 4 // SPECK uses 4 words (128 bits key)
#define SIMON_KEY_WORDS 4 // SIMON uses 4 words (128 bits key)

// SPECK Encryption Functions
void speck_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speck_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speck_round(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}
```

```
void speck_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    speck_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

// SIMON Encryption Functions
void simon_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}

void simon_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simon_round(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simon_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < rounds; i++)
  {
    simon_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
```

```
}

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Start the server
  server.begin();
  Serial.println("Server started.");
}

void loop()
{
  // Check if a client has connected
  WiFiClient client = server.available();

  if (client)
  {
    Serial.println("New Client connected.");

    // Choose encryption method
    bool use_speck = true; // Set to false to use SIMON

    char message[] = "Hello from ESP32 Server!";
    uint32_t plaintext[2], ciphertext[2];
    memcpy(plaintext, message, 8); // Encrypt the first 8 bytes for simplicity

    if (use_speck)
    {
      // SPECK Encryption
      uint8_t rounds = SPECK_ROUNDS_128;
      uint32_t key[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
      uint32_t key_schedule[rounds];

      speck_expand(key, key_schedule, rounds);
      speck_encrypt(plaintext, ciphertext, key_schedule, rounds);

      client.write((const uint8_t *)ciphertext, sizeof(ciphertext));
      Serial.println("Encrypted message sent with SPECK.");
    }
    else
    {
      // SIMON Encryption
      uint8_t rounds = SIMON_ROUNDS_128;
      uint32_t key[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
      uint32_t key_schedule[rounds];

      simon_expand(key, key_schedule, rounds);
      simon_encrypt(plaintext, ciphertext, key_schedule, rounds);

      client.write((const uint8_t *)ciphertext, sizeof(ciphertext));
      Serial.println("Encrypted message sent with SIMON.");
    }

    // Disconnect the client
    client.stop();
    Serial.println("Client disconnected.");
  }
}
```

6.4 Block cipher client-side code

```
#include <WiFi.h>
#include <stdint.h>

// Replace with your network credentials
const char *ssid = "*************";
const char *password = "*************";

// Server IP and Port (Replace with the server IP address and port)
const char *host = "192.168.1.10"; // Replace with your server's IP address
const uint16_t port = 80;

// Definitions for various encryption configurations
#define SPECK_ROUNDS_128 27 // Correct rounds for SPECK-128/128
#define SIMON_ROUNDS_128 44 // Correct rounds for SIMON-128/128
#define SPECK_KEY_WORDS 4   // SPECK uses 4 words (128 bits key)
#define SIMON_KEY_WORDS 4   // SIMON uses 4 words (128 bits key)

// SPECK Key Expansion
void speck_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    uint32_t &x = a[i % (SPECK_KEY_WORDS - 1)];
    b = (b >> 8) | (b << (32 - 8));
    b += x;
    b ^= i;
    x = (x << 3) | (x >> (32 - 3));
    x ^= b;
    S[i + 1] = b;
  }
}

// SPECK Decryption Functions
void speck_round_inv(uint32_t &x, uint32_t &y, uint32_t k)
{
  y ^= x;
  y = (y >> 3) | (y << (32 - 3));
  x ^= k;
  x -= y;
  x = (x << 8) | (x >> (32 - 8));
}

void speck_decrypt(uint32_t ciphertext[2], uint32_t plaintext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  plaintext[0] = ciphertext[0];
  plaintext[1] = ciphertext[1];
  for (i = rounds; i > 0; i--)
  {
    speck_round_inv(plaintext[1], plaintext[0], key_schedule[i - 1]);
  }
}

// SIMON Key Expansion
void simon_expand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    uint32_t &x = a[i % (SIMON_KEY_WORDS - 1)];
    uint32_t tmp = (b << 1) | (b >> (32 - 1));
    tmp &= (tmp << 8);
    tmp ^= x;
    x ^= i;
    x = (x >> 1) | (x << (32 - 1));
    S[i + 1] = b;
    b = tmp;
  }
}

// SIMON Decryption Functions
void simon_round_inv(uint32_t &x, uint32_t &y, uint32_t k)
{
  y ^= x;
  x ^= k;
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  y = tmp ^ y;
  x = (x >> 1) | (x << (32 - 1));
}
```

```
void simon_decrypt(uint32_t ciphertext[2], uint32_t plaintext[2], uint32_t
const key_schedule[], uint8_t rounds)
{
  uint32_t i;
  plaintext[0] = ciphertext[0];
  plaintext[1] = ciphertext[1];
  for (i = rounds; i > 0; i--)
  {
    simon_round_inv(plaintext[1], plaintext[0], key_schedule[i - 1]);
  }
}

void setup()
{
  Serial.begin(115200);

  // Connect to Wi-Fi network
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED)
  {
    delay(1000);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("WiFi connected.");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());

  // Connect to the server
  Serial.print("Connecting to ");
  Serial.print(host);
  Serial.print(":");
  Serial.println(port);

  WiFiClient client;

  if (client.connect(host, port))
  {
    Serial.println("Connected to server.");

    uint32_t ciphertext[2];
    int len = client.readBytes((char *)ciphertext, sizeof(ciphertext));
    if (len != sizeof(ciphertext))
    {
      Serial.println("Error: Did not receive expected ciphertext size.");
      return;
    }

    // Choose decryption method
    bool use_speck = true; // Set to false to use SIMON

    uint32_t plaintext[2];

    if (use_speck)
    {
      // SPECK Decryption
      uint8_t rounds = SPECK_ROUNDS_128;
      uint32_t key[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
      uint32_t key_schedule[rounds];

      speck_expand(key, key_schedule, rounds);
      speck_decrypt(ciphertext, plaintext, key_schedule, rounds);

      Serial.println("Decrypted message with SPECK:");
      Serial.println((char *)plaintext);
    }
    else
    {
      // SIMON Decryption
      uint8_t rounds = SIMON_ROUNDS_128;
      uint32_t key[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
      uint32_t key_schedule[rounds];

      simon_expand(key, key_schedule, rounds);
      simon_decrypt(ciphertext, plaintext, key_schedule, rounds);

      Serial.println("Decrypted message with SIMON:");
      Serial.println((char *)plaintext);
    }

    Serial.println("Disconnecting from server...");
    client.stop();
  }
  else
  {
    Serial.println("Connection to server failed.");
  }
}

void loop()
```

```
{
  // Add code here to periodically connect and communicate with the server
  delay(10000); // Wait 10 seconds between connections
}
```

Appendix 7: Approved Client List Process

```
// Server-side: Approved UUID list (this mimics immunological "whitelist")
std::vector<String> approvedUUIDs = {
    "UUID1-1234-5678-91011",
    "UUID2-9876-5432-10987",
    "UUID3-1357-2468-13579"};

// Function to check if a client is on the approved list
bool isClientApproved(const String &uuid)
{
  return std::find(approvedUUIDs.begin(), approvedUUIDs.end(), uuid) !=
approvedUUIDs.end();
}

// Handler invoked on incoming client connection
void handleClientConnection(WiFiClient &client)
{
  String receivedUUID = client.readStringUntil('\n');

  if (isClientApproved(receivedUUID))
  {
    Serial.println("Client UUID approved.");
    // Proceed with session setup, encryption negotiation, etc.
  }
  else
  {
    Serial.println("Rejected client – UUID not approved.");
    client.println("Access Denied: UUID not recognised.");
    client.stop();
  }
}
```

This process implements the first layer of access control in the security framework. When a client attempts to connect, its UUID is checked against a pre-defined list of authorised identifiers. This logic, inspired by immunological self/non-self recognition, ensures that only trusted nodes proceed to encryption negotiation and session setup.

Appendix 8: Client Connection Logic

```
// Function to handle session setup based on client UUID and battery state
void establishClientSession(const String& clientUUID, float batteryPercentage)
{
  Session newSession;
  newSession.clientUUID = clientUUID;
  newSession.isApproved = true;
  newSession.sessionExpirationTime = millis() +
determineSessionDuration(batteryPercentage);
  newSession.isSleeping = false;
  sessions.push_back(newSession);

  Serial.print("New session for ");
  Serial.print(clientUUID);
  Serial.print(" with expiration in ");
  Serial.print(newSession.sessionExpirationTime);
  Serial.println(" ms");
}
```

This function establishes the connection lifecycle for each client based on its UUID and energy profile. It supports dynamic expiry durations, mapped through the battery curve logic.

Appendix 9: Shared Ledger Process

```
// Server-side function to broadcast ledger changes
void notifyApprovedClientsOfLedgerUpdate()
{
  for (auto& session : sessions) {
    if (session.isApproved) {
      sendLedgerToClient(session.clientUUID);
    }
  }
}

// Client-side function to receive updated credentials
void updateLocalLedger(const LedgerEntry& newEntry)
{
  currentLedgerEntry = newEntry;
  Serial.println("Local ledger updated successfully.");
}
```

These routines govern synchronisation of encryption credentials across all approved clients using a blockchain-inspired "broadcast and adopt" pattern.

Appendix 10: Auto Detection System

```
bool isClientApproved(const String& uuid)
{
  return std::find(approvedUUIDs.begin(), approvedUUIDs.end(), uuid) !=
approvedUUIDs.end();
}

bool isClientDesynced(const String& ssid, const String& password)
{
  return ssid != currentLedgerEntry.ssid || password !=
currentLedgerEntry.password;
}

void processConnectionRequest(const String& uuid, const String& ssid, const
String& password)
{
  if (!isClientApproved(uuid))
  {
    rejectConnection("Unapproved UUID");
    return;
  }

  if (isClientDesynced(ssid, password))
  {
    rejectConnection("Desynchronisation detected");
    return;
  }

  Serial.println("Client approved and in sync.");
}
```

This logic mirrors immunological surveillance, checking UUIDs and credentials against a central reference set.

Appendix 11: Auto Ejection System

```
void handleCompromisedCredentials(const String &clientUUID)
{
  removeSession(clientUUID);
  updateServerCredentials();
  notifyApprovedClientsOfLedgerUpdate();

  Serial.print("Ejected client: ");
  Serial.println(clientUUID);
}
```

This mechanism removes misaligned or unapproved sessions and updates all other participants, akin to a cytotoxic immune response.

Appendix 12: Trigger System

```
void triggerCredentialUpdate()
{
  currentLedgerIndex = (currentLedgerIndex + 1) % MAX_LEDGER_ENTRIES;
  updateServerCredentials();
  notifyApprovedClientsOfLedgerUpdate();
}
```

This function initiates a full-system immune memory response, rotating encryption credentials and propagating them to all verified clients after a security event.

Appendix 13: Cycle Testing For Encryption Methods

13.1 Cycle Function for Xtensa Processors

```
uint64_t read_cycles()
{
  return (uint64_t)esp_cpu_get_ccount(); // Returns current CPU cycle count
}
```

13.2 XOR Encryption

```
void xor_encryption()
{
  char key = 0xAA; // Simple XOR key

  char data_copy[sizeof(data)];
  strcpy(data_copy, data);

  start_time = micros();
  start_cycles = read_cycles();

  for (int i = 0; i < sizeof(data_copy) - 1; i++)
  {
    data_copy[i] ^= key;
  }

  end_cycles = read_cycles();
  end_time = micros();

  Serial.println("XOR Encryption:");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
  Serial.print("Cycles taken: ");
  Serial.println(end_cycles - start_cycles);
}
```

13.3 Caesar Cipher

```
void caesar_cipher()
{
  int shift = 3; // Caesar Cipher shift

  char data_copy[sizeof(data)];
  strcpy(data_copy, data);

  start_time = micros();
  start_cycles = read_cycles();

  for (int i = 0; i < sizeof(data_copy) - 1; i++)
  {
    if (isalpha(data_copy[i]))
    {
      char offset = isupper(data_copy[i]) ? 'A' : 'a';
      data_copy[i] = (data_copy[i] - offset + shift) % 26 + offset;
    }
  }

  end_cycles = read_cycles();
  end_time = micros();

  Serial.println("Caesar Cipher:");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
  Serial.print("Cycles taken: ");
  Serial.println(end_cycles - start_cycles);
}
```

13.4 ROT13 Encryption

```
void rot13_encryption()
{
  char data_copy[sizeof(data)];
  strcpy(data_copy, data);

  start_time = micros();
  start_cycles = read_cycles();

  for (int i = 0; i < sizeof(data_copy) - 1; i++)
  {
    if (isalpha(data_copy[i]))
    {
      char offset = isupper(data_copy[i]) ? 'A' : 'a';
      data_copy[i] = (data_copy[i] - offset + 13) % 26 + offset;
    }
  }

  end_cycles = read_cycles();
  end_time = micros();

  Serial.println("ROT13 Encryption:");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
  Serial.print("Cycles taken: ");
  Serial.println(end_cycles - start_cycles);
}
```

13.5 SPECK Encryption

```
void speck_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speck_expand(uint32_t const K[SPECK_KEY_WORDS], uint32_t S[SPECK_ROUNDS])
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];

  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }

  S[0] = b;
  for (i = 0; i < SPECK_ROUNDS - 1; i++)
  {
    speck_round(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speck_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[SPECK_ROUNDS])
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < SPECK_ROUNDS; i++)
  {
    speck_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

void speck_encryption()
{
  uint32_t key[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
  uint32_t key_schedule[SPECK_ROUNDS];
  uint32_t plaintext[2], ciphertext[2];

  speck_expand(key, key_schedule);
  start_time = micros();
  start_cycles = read_cycles();

  for (int i = 0; i < sizeof(data) - 1; i += 8)
  {
    memcpy(plaintext, data + i, 8);
    speck_encrypt(plaintext, ciphertext, key_schedule);
  }

  end_cycles = read_cycles();
  end_time = micros();

  Serial.println("SPECK Encryption:");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
  Serial.print("Cycles taken: ");
  Serial.println(end_cycles - start_cycles);
}
```

13.6 SIMON Encryption

```
void simon_round(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}

void simon_expand(uint32_t const K[SIMON_KEY_WORDS], uint32_t S[SIMON_ROUNDS])
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];

  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }

  S[0] = b;
  for (i = 0; i < SIMON_ROUNDS - 1; i++)
  {
    simon_round(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simon_encrypt(uint32_t plaintext[2], uint32_t ciphertext[2], uint32_t
const key_schedule[SIMON_ROUNDS])
{
  uint32_t i;
  ciphertext[0] = plaintext[0];
  ciphertext[1] = plaintext[1];
  for (i = 0; i < SIMON_ROUNDS; i++)
  {
    simon_round(ciphertext[1], ciphertext[0], key_schedule[i]);
  }
}

void simon_encryption()
{
  uint32_t key[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
  uint32_t key_schedule[SIMON_ROUNDS];
  uint32_t plaintext[2], ciphertext[2];

  simon_expand(key, key_schedule);
  start_time = micros();
  start_cycles = read_cycles();

  for (int i = 0; i < sizeof(data) - 1; i += 8)
  {
    memcpy(plaintext, data + i, 8);
    simon_encrypt(plaintext, ciphertext, key_schedule);
  }

  end_cycles = read_cycles();
  end_time = micros();

  Serial.println("SIMON Encryption:");
  Serial.print("Time taken (microseconds): ");
  Serial.println(end_time - start_time);
  Serial.print("Cycles taken: ");
  Serial.println(end_cycles - start_cycles);
}
```

Appendix 14: Adaptive Amoeba Battery Curve Mapping Management System

```
#include <WiFi.h>
#include <EEPROM.h>

#define ADC_PIN A0 // Adjust to device configured ADC pin
#define EEPROM_SIZE 512

#define MAX_DATA_POINTS 100

struct BatteryData
{
  uint32_t voltage;
  float percentage;
};

BatteryData batteryData[MAX_DATA_POINTS];
int dataPointsCount = 0;

uint32_t getBatteryVoltage()
{
  int adcValue = analogRead(ADC_PIN);
  // Convert ADC value to voltage (ESP32 ADC is 12-bit, 4095 max value)
  uint32_t voltage = map(adcValue, 0, 4095, 0, 5000); // Adjust according to
your ADC reference voltage
  return voltage;
}

float initialMapBatteryCurve(uint32_t voltage)
{
  if (voltage > 4200)
    return 100.0;
  else if (voltage > 4000)
    return 75.0 + (voltage - 4000) * 0.25;
  else if (voltage > 3800)
    return 50.0 + (voltage - 3800) * 0.125;
  else if (voltage > 3600)
    return 25.0 + (voltage - 3600) * 0.125;
  else if (voltage > 3400)
    return (voltage - 3400) * 0.125;
  else
    return 0.0;
}

// Non-linear interpolation for more accurate prediction
float nonLinearInterpolation(uint32_t voltage)
{
  float voltageNormalized = (float)(voltage - 3400) / (4200 - 3400); //
Normalize voltage
  return pow(voltageNormalized, 2) * 100.0;                          //
Quadratic non-linear mapping
}

// Dynamic EEPROM Management
void storeBatteryData(uint32_t voltage, float percentage)
{
  if (dataPointsCount < MAX_DATA_POINTS)
  {
    // Add data until max capacity is reached
    batteryData[dataPointsCount].voltage = voltage;
    batteryData[dataPointsCount].percentage = percentage;
    dataPointsCount++;
  }
  else
  {
    // Overwrite oldest data Circular Buffer Approach
    for (int i = 1; i < MAX_DATA_POINTS; i++)
    {
      batteryData[i - 1] = batteryData[i];
    }
    batteryData[MAX_DATA_POINTS - 1].voltage = voltage;
    batteryData[MAX_DATA_POINTS - 1].percentage = percentage;
  }

  EEPROM.put(0, batteryData);
  EEPROM.put(sizeof(batteryData), dataPointsCount);
  EEPROM.commit();
}

// Predictive Maintenance and Battery Health Estimation
float estimateBatteryHealth(uint32_t voltage)
{
  // Simplistic battery health estimation based on voltage range
  if (voltage > 4200)
    return 1.0; // Healthy battery
  else if (voltage > 4000)
    return 0.75;
  else if (voltage > 3800)
    return 0.5;
  else if (voltage > 3600)
    return 0.25;
  else
    return 0.1; // Battery near end of life
}
```

```
void loadBatteryData()
{
  EEPROM.get(0, batteryData);
  EEPROM.get(sizeof(batteryData), dataPointsCount);
}

float predictBatteryPercentage(uint32_t voltage)
{
  if (dataPointsCount < 2)
  {
    return initialMapBatteryCurve(voltage);
  }

  // Apply non-linear interpolation using collected data
  return nonLinearInterpolation(voltage);
}

// Adaptive Sleep and Wake-Up Intervals
void enterAdaptiveSleepMode(float batteryPercentage, float batteryHealth)
{
  uint64_t sleepDuration;

  if (batteryPercentage > 75.0)
  {
    Serial.println("Battery sufficient, normal operation.");
    sleepDuration = 10000000; // Normal operation with regular wake-up
  }
  else if (batteryPercentage > 50.0)
  {
    Serial.println("Entering light sleep mode.");
    sleepDuration = batteryHealth * 20000000; // Sleep duration adapted based
on battery health
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_light_sleep_start();
  }
  else if (batteryPercentage > 25.0)
  {
    Serial.println("Entering deep sleep mode.");
    sleepDuration = batteryHealth * 40000000; // Longer sleep duration for
lower battery health
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_deep_sleep_start();
  }
  else
  {
    Serial.println("Entering ULP mode, maximizing battery life.");
    sleepDuration = batteryHealth * 60000000; // Maximize sleep duration in
ULP mode
```

```
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_deep_sleep_start();
  }
}

void setup()
{
  Serial.begin(115200);
  EEPROM.begin(EEPROM_SIZE);

  loadBatteryData(); // Load stored battery data

  Serial.println("Starting battery management system...");
}

void loop()
{
  uint32_t voltage = getBatteryVoltage();
  float batteryPercentage = predictBatteryPercentage(voltage);
  float batteryHealth = estimateBatteryHealth(voltage);

  Serial.printf("Battery Voltage: %d mV, Predicted Battery Percentage: %.2f%%, 
Estimated Battery Health: %.2f\n", voltage, batteryPercentage, batteryHealth);

  storeBatteryData(voltage, batteryPercentage); // Store the new data point

  enterAdaptiveSleepMode(batteryPercentage, batteryHealth); // Enter 
appropriate adaptive sleep mode

  delay(10000); // Wait for 10 seconds before the next reading
}
```

Appendix 15: Full System Without Adaptive Amoeba Battery Curve Mapping Management System

15.1 Server

```
#include <WiFi.h>
#include <stdint.h>

// Constants and Macros
#define SERVER_PORT 80

#define SPECK_ROUNDS_128 22
#define SIMON_ROUNDS_128 44

#define SPECK_KEY_WORDS 4
#define SIMON_KEY_WORDS 4

// Structure Definitions
struct ClientSession
{
  String sessionID;
  String clientUUID;
  bool approved;
  WiFiClient clientConnection;
  bool reconnected;
  unsigned long lastActive;
};

struct LedgerEntry
{
  String ssid;
  String password;
  String encryptionKey;
  String encryptionMethod;
};

// Global Variables
WiFiServer server(SERVER_PORT);
ClientSession sessions[10];
int sessionCount = 0;
LedgerEntry ledger[5] = {
    {"SSID1", "Password1", "XORKey", "XOR"},
    {"SSID2", "Password2", "CaesarKey", "Caesar"},
    {"SSID3", "Password3", "ROT13Key", "ROT13"},
    {"SSID4", "Password4", "SpeckKey", "SPECK"},
    {"SSID5", "Password5", "SimonKey", "SIMON"}};
int currentLedgerIndex = 0;
unsigned long sessionExpiryTime = 120000; // Session expires after 120 seconds
of inactivity
```

```
// Honeypot SSID and Password
String honeypotSSID = "Honeypot_SSID";
String honeypotPassword = "Honeypot_Password";
bool honeypotActive = false;

// List of approved UUIDs (dummy data)
String approvedUUIDs[] = {
    "UUID1-1234-5678-91011",
    "UUID2-1234-5678-91011",
    "UUID3-1234-5678-91011"};
bool uuidConnected[] = {false, false, false};

int approvedUUIDCount = sizeof(approvedUUIDs) / sizeof(approvedUUIDs[0]);

// XOR Encryption/Decryption
void xorEncryptionDecryption(char *data, String key)
{
  for (int i = 0; i < strlen(data); i++)
  {
    data[i] ^= key[0];
  }
}

// Caesar Cipher Encryption/Decryption
void caesarCipherEncryptionDecryption(char *data, int shift)
{
  for (int i = 0; i < strlen(data); i++)
  {
    if (isalpha(data[i]))
    {
      char offset = isupper(data[i]) ? 'A' : 'a';
      data[i] = (data[i] - offset + shift) % 26 + offset;
    }
  }
}

// ROT13 Encryption/Decryption
void rot13EncryptionDecryption(char *data)
{
  caesarCipherEncryptionDecryption(data, 13);
}

// SPECK Encryption/Decryption
void speckRound(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speckExpand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speckRound(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speckEncryptDecrypt(uint32_t data[2], uint32_t const key_schedule[],
uint8_t rounds)
{
  for (uint8_t i = 0; i < rounds; i++)
  {
    speckRound(data[1], data[0], key_schedule[i]);
  }
}

// SIMON Encryption/Decryption
void simonRound(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}

void simonExpand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simonRound(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simonEncryptDecrypt(uint32_t data[2], uint32_t const key_schedule[],
uint8_t rounds)
{
  for (uint8_t i = 0; i < rounds; i++)
  {
    simonRound(data[1], data[0], key_schedule[i]);
  }
}

void encryptDecryptData(char *data, const String &method, const String &key)
{
  if (method == "XOR")
  {
    xorEncryptionDecryption(data, key);
  }
  else if (method == "Caesar")
  {
    caesarCipherEncryptionDecryption(data, key.toInt());
  }
  else if (method == "ROT13")
  {
    rot13EncryptionDecryption(data);
  }
  else if (method == "SPECK")
  {
    uint32_t key_schedule[SPECK_ROUNDS_128];
    uint32_t key_words[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
    speckExpand(key_words, key_schedule, SPECK_ROUNDS_128);
    uint32_t block[2];
    memcpy(block, data, 8);
    speckEncryptDecrypt(block, key_schedule, SPECK_ROUNDS_128);
    memcpy(data, block, 8);
  }
  else if (method == "SIMON")
  {
    uint32_t key_schedule[SIMON_ROUNDS_128];
    uint32_t key_words[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
    simonExpand(key_words, key_schedule, SIMON_ROUNDS_128);
    uint32_t block[2];
    memcpy(block, data, 8);
    simonEncryptDecrypt(block, key_schedule, SIMON_ROUNDS_128);
    memcpy(data, block, 8);
  }
}

// Function to generate a session ID
String generateSessionID()
{
  String sessionID = "";
  for (int i = 0; i < 16; i++)
  {
    sessionID += String(random(0, 16), HEX);
  }
  delay(100); // Prevents immediate reuse of session ID
  return sessionID;
}

// Function to check if a session ID is approved
bool isSessionApproved(String sessionID, String clientUUID)
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].sessionID == sessionID && sessions[i].clientUUID ==
clientUUID && sessions[i].approved)
    {
      return true;
    }
  }
  return false;
}

// Function to check if a UUID is approved
bool isUUIDApproved(String uuid)
{
  for (int i = 0; i < approvedUUIDCount; i++)
  {
    if (approvedUUIDs[i] == uuid)
    {
      return true;
    }
  }
  return false;
}

// Function to update the server's SSID and password based on the current
ledger entry
```

```
void updateServerCredentials()
{
  WiFi.softAPdisconnect(true);
  WiFi.softAP(ledger[currentLedgerIndex].ssid.c_str(),
ledger[currentLedgerIndex].password.c_str());
  Serial.println("Server SSID and Password updated to:");
  Serial.println("SSID: " + ledger[currentLedgerIndex].ssid);
  Serial.println("Password: " + ledger[currentLedgerIndex].password);
}

// Function to start the honeypot SSID
void startHoneypot()
{
  WiFi.softAPdisconnect(true);
  WiFi.softAP(honeypotSSID.c_str(), honeypotPassword.c_str());
  honeypotActive = true;
  Serial.println("Honeypot SSID and Password activated:");
  Serial.println("Honeypot SSID: " + honeypotSSID);
  Serial.println("Honeypot Password: " + honeypotPassword);
}

// Function to stop the honeypot SSID
void stopHoneypot()
{
  WiFi.softAPdisconnect(true);
  honeypotActive = false;
  updateServerCredentials();
  Serial.println("Honeypot SSID deactivated. Server SSID and Password
restored.");
}

// Function to notify all approved clients of a ledger update
void notifyApprovedClients()
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].approved && sessions[i].clientConnection.connected())
    {
      sessions[i].clientConnection.println("Ledger Update");
      sessions[i].clientConnection.println(currentLedgerIndex);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].ssid);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].password
);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].encrypti
onKey);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].encrypti
onMethod);
    }
  }
}

// Function to handle compromised credentials and trigger process
void handleCompromisedCredentials(String sessionID, String clientUUID)
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].sessionID == sessionID && sessions[i].clientUUID ==
clientUUID)
    {
      Serial.println("Session ID " + sessionID + " marked as compromised.");
      sessions[i].approved = false;
      currentLedgerIndex = (currentLedgerIndex + 1) % 5;
      updateServerCredentials();
      notifyApprovedClients();
      startHoneypot();
      break;
    }
  }
  for (int i = 0; i < sessionCount; i++)
  {
    sessions[i].reconnected = false;
  }
  Serial.println("All sessions marked as needing reconnection.");
}

// Function to check for desynchronization and send error code (Only for new
UUID connections)
void checkDesynchronization(WiFiClient &client, String clientUUID)
{
  for (int i = 0; i < approvedUUIDCount; i++)
  {
    if (approvedUUIDs[i] == clientUUID && !uuidConnected[i])
    {
      String clientSSID = client.readStringUntil('\n');
      String clientPassword = client.readStringUntil('\n');

      if (clientSSID != ledger[currentLedgerIndex].ssid || clientPassword !=
ledger[currentLedgerIndex].password)
      {
        Serial.println("Desync detected for UUID: " + clientUUID);
        client.println("Error: Desync detected. Please reconnect.");
        client.stop();
        return;
      }
      uuidConnected[i] = true;
      Serial.println("UUID " + clientUUID + " connected successfully.");
    }
  }
}

// Function to handle new incoming connections (Server-side)
void handleNewConnection(WiFiClient client)
{
  String sessionID = generateSessionID();
  client.println(sessionID);

  String clientInfo = client.readStringUntil('\n');
  String clientUUID = client.readStringUntil('\n');
  String receivedSessionID = client.readStringUntil('\n');

  String encryptionMethod = ledger[currentLedgerIndex].encryptionMethod;
  client.println(encryptionMethod);

  if (isUUIDApproved(clientUUID))
  {
    if (!isSessionApproved(receivedSessionID, clientUUID))
    {
      sessions[sessionCount].sessionID = sessionID;
      sessions[sessionCount].clientUUID = clientUUID;
      sessions[sessionCount].approved = true;
      sessions[sessionCount].clientConnection = client;
      sessions[sessionCount].reconnected = false;
      sessions[sessionCount].lastActive = millis();
      sessionCount++;

      Serial.println("Session ID " + sessionID + " for UUID " + clientUUID + "
connected.");

      checkDesynchronization(client, clientUUID);

      client.println("Connection Successful");

      client.println(ledger[currentLedgerIndex].ssid);
      client.println(ledger[currentLedgerIndex].password);
      client.println(ledger[currentLedgerIndex].encryptionKey);
      client.println(ledger[currentLedgerIndex].encryptionMethod);

      for (int i = 0; i < sessionCount; i++)
      {
        if (sessions[i].clientConnection == client)
        {
          sessions[i].reconnected = true;
          Serial.println("Session ID " + sessionID + " marked as
reconnected.");
```

```
          break;
        }
      }
    }
    else
    {
      Serial.println("Session ID " + receivedSessionID + " is already 
approved. Skipping reinitialization.");
      client.println("Connection Successful");

      client.println(ledger[currentLedgerIndex].ssid);
      client.println(ledger[currentLedgerIndex].password);
      client.println(ledger[currentLedgerIndex].encryptionKey);
      client.println(ledger[currentLedgerIndex].encryptionMethod);
    }
  }
  else
  {
    Serial.println("Failed connection attempt with Session ID " + sessionID + 
" and UUID " + clientUUID);
    client.println("Session ID blocked or compromised");
    client.stop();
  }

  String sensorData = client.readStringUntil('\n');
  encryptDecryptData(&sensorData[0], 
ledger[currentLedgerIndex].encryptionMethod, 
ledger[currentLedgerIndex].encryptionKey);
  Serial.println("Received Decrypted Sensor Data: " + sensorData);
  client.println("Sensor data received and decrypted successfully.");

  bool allReconnected = true;
  for (int i = 0; i < approvedUUIDCount; i++)
  {
    if (!uuidConnected[i])
    {
      allReconnected = false;
      Serial.println("UUID " + approvedUUIDs[i] + " has not reconnected 
yet.");
      break;
    }
  }
  if (allReconnected)
  {
    Serial.println("All approved UUIDs have reconnected.");
    if (honeypotActive)
    {
      stopHoneypot();
    }
  }
}

// Cleanup expired sessions
void cleanupExpiredSessions()
{
  unsigned long currentTime = millis();
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].approved && (currentTime - sessions[i].lastActive >
sessionExpiryTime))
    {
      Serial.println("Session ID " + sessions[i].sessionID + " expired due to
inactivity.");
      sessions[i].approved = false;
      sessions[i].clientConnection.stop(); // Close the connection to the
expired session
    }
  }
}

// Setup function (Server-side)
void setup()
{
  Serial.begin(115200);
  WiFi.softAP(ledger[currentLedgerIndex].ssid.c_str(),
ledger[currentLedgerIndex].password.c_str());
  server.begin();

  Serial.println("Server started with initial SSID and Password:");
  Serial.println("SSID: " + ledger[currentLedgerIndex].ssid);
  Serial.println("Password: " + ledger[currentLedgerIndex].password);
}

// Main loop (Server-side)
void loop()
{
  WiFiClient client = server.available();
  if (client)
  {
    handleNewConnection(client);
  }

  cleanupExpiredSessions(); // Cleanup expired sessions

  delay(2000); // Delay for demonstration purposes
}
```

15.2 Client

```
#include <WiFi.h>
#include <stdint.h>

// Server details
const char *host = "192.168.4.1"; // IP address of the server
const uint16_t port = 80;

// Constants for encryption
#define SPECK_ROUNDS_128 22
#define SIMON_ROUNDS_128 44

#define SPECK_KEY_WORDS 4
#define SIMON_KEY_WORDS 4

WiFiClient client;

// Ledger entries to store received data
struct LedgerEntry
{
  String ssid;
  String password;
  String encryptionKey;
  String encryptionMethod;
};

LedgerEntry storedLedger[5];
int storedLedgerCount = 0;

// Honeypot SSID details
const char *honeypotSSID = "Honeypot_SSID";
const char *honeypotPassword = "Honeypot_Password";

// Sample data to be encrypted and sent
char data[] = "The quick brown fox jumps over the lazy dog
1234567890!@#$%^&*()_+-=[]{}|;':,.<>/?\n\t";

// XOR Encryption/Decryption
void xorEncryptionDecryption(char *data, String key)
{
  for (int i = 0; i < strlen(data); i++)
  {
    data[i] ^= key[0];
  }
}

// Caesar Cipher Encryption/Decryption
void caesarCipherEncryptionDecryption(char *data, int shift)
```

```
{
  for (int i = 0; i < strlen(data); i++)
  {
    if (isalpha(data[i]))
    {
      char offset = isupper(data[i]) ? 'A' : 'a';
      data[i] = (data[i] - offset + shift) % 26 + offset;
    }
  }
}

// ROT13 Encryption/Decryption
void rot13EncryptionDecryption(char *data)
{
  caesarCipherEncryptionDecryption(data, 13);
}

// SPECK Encryption/Decryption
void speckRound(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speckExpand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speckRound(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speckEncryptDecrypt(uint32_t data[2], uint32_t const key_schedule[],
uint8_t rounds)
{
  for (uint8_t i = 0; i < rounds; i++)
  {
    speckRound(data[1], data[0], key_schedule[i]);
  }
}

// SIMON Encryption/Decryption
void simonRound(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}

void simonExpand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simonRound(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simonEncryptDecrypt(uint32_t data[2], uint32_t const key_schedule[],
uint8_t rounds)
{
  for (uint8_t i = 0; i < rounds; i++)
  {
    simonRound(data[1], data[0], key_schedule[i]);
  }
}

void encryptDecryptData(char *data, const String &method, const String &key)
{
  if (method == "XOR")
  {
    xorEncryptionDecryption(data, key);
  }
  else if (method == "Caesar")
  {
    caesarCipherEncryptionDecryption(data, key.toInt());
  }
  else if (method == "ROT13")
  {
    rot13EncryptionDecryption(data);
  }
  else if (method == "SPECK")
  {
    uint32_t key_schedule[SPECK_ROUNDS_128];
    uint32_t key_words[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
    speckExpand(key_words, key_schedule, SPECK_ROUNDS_128);
    uint32_t block[2];
    memcpy(block, data, 8);
    speckEncryptDecrypt(block, key_schedule, SPECK_ROUNDS_128);
    memcpy(data, block, 8);
  }
  else if (method == "SIMON")
  {
    uint32_t key_schedule[SIMON_ROUNDS_128];
    uint32_t key_words[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
    simonExpand(key_words, key_schedule, SIMON_ROUNDS_128);
    uint32_t block[2];
    memcpy(block, data, 8);
    simonEncryptDecrypt(block, key_schedule, SIMON_ROUNDS_128);
    memcpy(data, block, 8);
  }
}

bool connectToWiFi(const char *ssid, const char *password)
{
  WiFi.begin(ssid, password);

  int attempts = 0;
  while (WiFi.status() != WL_CONNECTED && attempts < 20)
  {
    delay(500);
    Serial.print(".");
    attempts++;
  }

  if (WiFi.status() == WL_CONNECTED)
  {
    Serial.println("\nConnected to Wi-Fi");
    return true;
  }

  Serial.println("\nFailed to connect to Wi-Fi");
  return false;
}

bool connectToWiFiWithStoredLedger()
{
  if (storedLedgerCount > 0)
  {
    return connectToWiFi(storedLedger[storedLedgerCount - 1].ssid.c_str(),
storedLedger[storedLedgerCount - 1].password.c_str());
  }
  return false;
}

void handleDesync()
{
  Serial.println("Handling desync, attempting to connect to honeypot...");

  if (connectToWiFi(honeypotSSID, honeypotPassword))
  {
    if (client.connect(host, port))
    {
      Serial.println("Connected to server via honeypot");

      String uuid = "UUID1-1234-5678-91011";
      client.println(uuid);

      for (int i = 0; i < 5; i++)
      {
        storedLedger[i].ssid = client.readStringUntil('\n');
        storedLedger[i].password = client.readStringUntil('\n');
        storedLedger[i].encryptionKey = client.readStringUntil('\n');
        storedLedger[i].encryptionMethod = client.readStringUntil('\n');
        storedLedgerCount++;
      }

      Serial.println("Updated ledger received:");
      for (int i = 0; i < storedLedgerCount; i++)
      {
        Serial.println("SSID: " + storedLedger[i].ssid);
        Serial.println("Password: " + storedLedger[i].password);
        Serial.println("Encryption Key: " + storedLedger[i].encryptionKey);
        Serial.println("Encryption Method: " +
storedLedger[i].encryptionMethod);
      }

      if (connectToWiFiWithStoredLedger())
      {
        Serial.println("Reconnected to server using updated ledger
credentials");
      }
      else
      {
        Serial.println("Failed to reconnect to server after receiving updated
ledger");
      }
    }
    else
    {
      Serial.println("Failed to connect to server via honeypot");
    }
  }
  else
  {
    Serial.println("Failed to connect to honeypot SSID");
  }
}

void handleLedgerUpdate() {
    if (client.connected()) {
        String ledgerUpdate = client.readStringUntil('\n');
        if (ledgerUpdate == "Ledger Update") {
            int newIndex = client.readStringUntil('\n').toInt();
            storedLedger[newIndex].ssid = client.readStringUntil('\n');
            storedLedger[newIndex].password = client.readStringUntil('\n');
            storedLedger[newIndex].encryptionKey =
client.readStringUntil('\n');
            storedLedger[newIndex].encryptionMethod =
client.readStringUntil('\n');

            Serial.println("Ledger updated:");
            Serial.println("New SSID: " + storedLedger[newIndex].ssid);
            Serial.println("New Password: " +
storedLedger[newIndex].password);
            Serial.println("New Encryption Key: " +
storedLedger[newIndex].encryptionKey);
            Serial.println("New Encryption Method: " +
storedLedger[newIndex].encryptionMethod);

            if (!connectToWiFiWithStoredLedger()) {
                Serial.println("Error: Desynced from server. Unable to
reconnect.");
                handleDesync();
            }
        } else {
            Serial.println("Unexpected response during ledger update: " +
ledgerUpdate);
        }
    } else {
        Serial.println("Client disconnected unexpectedly during ledger
update.");
    }
}

void setup()
{
  Serial.begin(115200);

  if (!connectToWiFiWithStoredLedger())
  {
    Serial.println("Error: Initial connection failed. Desynced from server.");
    handleDesync();
    return;
  }

  if (client.connect(host, port))
  {
    Serial.println("Connected to server");

    String sessionID = client.readStringUntil('\n');
    Serial.println("Received Session ID: " + sessionID);

    String processingPower = "HIGH";
    String uuid = "UUID1-1234-5678-91011";
    client.println(processingPower);
    client.println(uuid);
    client.println(sessionID);

    String encryptionMethod = client.readStringUntil('\n');
    Serial.println("Encryption Method: " + encryptionMethod);

    String response = client.readStringUntil('\n');
    Serial.println("Server Response: " + response);

    if (response == "Connection Successful")
    {
      for (int i = 0; i < 5; i++)
      {
        storedLedger[i].ssid = client.readStringUntil('\n');
        storedLedger[i].password = client.readStringUntil('\n');
        storedLedger[i].encryptionKey = client.readStringUntil('\n');
        storedLedger[i].encryptionMethod = client.readStringUntil('\n');
        storedLedgerCount++;
      }

      Serial.println("Received full ledger:");
      for (int i = 0; i < storedLedgerCount; i++)
      {
        Serial.println("SSID: " + storedLedger[i].ssid);
        Serial.println("Password: " + storedLedger[i].password);
        Serial.println("Encryption Key: " + storedLedger[i].encryptionKey);
        Serial.println("Encryption Method: " +
storedLedger[i].encryptionMethod);
      }

      encryptDecryptData(data, encryptionMethod,
storedLedger[0].encryptionKey);
      client.println(data);

      String confirmation = client.readStringUntil('\n');
      Serial.println("Server Confirmation: " + confirmation);
    }
  }
  else
  {
    Serial.println("Failed to connect to server");
  }
}

void loop()
{
  if (client.connected())
  {
    handleLedgerUpdate();
  }
  else
  {
    Serial.println("Disconnected from server");

    if (connectToWiFiWithStoredLedger() && client.connect(host, port))
    {
      Serial.println("Reconnected to server using ledger credentials");
    }
    else
    {
      Serial.println("Failed to reconnect to server");
      handleDesync();
    }
  }
  delay(5000);
}
```

Appendix 16: Full System With Adaptive Amoeba Battery Curve Mapping Management System

16.1 Server

```
#include <WiFi.h>
#include <stdint.h>

// Constants and Macros
#define SERVER_PORT 80

#define SPECK_ROUNDS_128 22
#define SIMON_ROUNDS_128 44
#define SPECK_KEY_WORDS 4
#define SIMON_KEY_WORDS 4

// Structure Definitions
struct ClientSession
{
  String sessionID;
  String clientUUID;
  bool approved;
  WiFiClient clientConnection;
  bool reconnected;
  uint32_t sessionExpirationTime;
  bool isSleeping;
};

struct LedgerEntry
{
  String ssid;
  String password;
  String encryptionKey;
  String encryptionMethod;
};

// Global Variables
WiFiServer server(SERVER_PORT);
ClientSession sessions[10];
int sessionCount = 0;
LedgerEntry ledger[5] = {
    {"SSID1", "Password1", "XORKey", "XOR"},
    {"SSID2", "Password2", "CaesarKey", "Caesar"},
    {"SSID3", "Password3", "ROT13Key", "ROT13"},
    {"SSID4", "Password4", "SpeckKey", "SPECK"},
    {"SSID5", "Password5", "SimonKey", "SIMON"}};
int currentLedgerIndex = 0;

String approvedUUIDs[] = {
    "UUID1-1234-5678-91011",
    "UUID2-1234-5678-91011",
    "UUID3-1234-5678-91011"};
bool uuidConnected[] = {false, false, false};

int approvedUUIDCount = sizeof(approvedUUIDs) / sizeof(approvedUUIDs[0]);

// XOR Encryption/Decryption
void xorEncryptionDecryption(char *data, String key)
{
  for (int i = 0; i < strlen(data); i++)
  {
    data[i] ^= key[0];
  }
}

// Caesar Cipher Encryption/Decryption
void caesarCipherEncryptionDecryption(char *data, int shift)
{
  for (int i = 0; i < strlen(data); i++)
  {
    if (isalpha(data[i]))
    {
      char offset = isupper(data[i]) ? 'A' : 'a';
      data[i] = (data[i] - offset + shift) % 26 + offset;
    }
  }
}

// ROT13 Encryption/Decryption
void rot13EncryptionDecryption(char *data)
{
  caesarCipherEncryptionDecryption(data, 13);
}

// SPECK Encryption/Decryption
void speckRound(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speckExpand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speckRound(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speckEncryptDecrypt(uint32_t data[2], uint32_t const key_schedule[],
uint8_t rounds)
{
  for (uint8_t i = 0; i < rounds; i++)
  {
    speckRound(data[1], data[0], key_schedule[i]);
  }
}

// SIMON Encryption/Decryption
void simonRound(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}

void simonExpand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simonRound(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}
```

```
void simonEncryptDecrypt(uint32_t data[2], uint32_t const key_schedule[],
uint8_t rounds)
{
  for (uint8_t i = 0; i < rounds; i++)
  {
    simonRound(data[1], data[0], key_schedule[i]);
  }
}

void encryptDecryptData(char *data, const String &method, const String &key)
{
  if (method == "XOR")
  {
    xorEncryptionDecryption(data, key);
  }
  else if (method == "Caesar")
  {
    caesarCipherEncryptionDecryption(data, key.toInt());
  }
  else if (method == "ROT13")
  {
    rot13EncryptionDecryption(data);
  }
  else if (method == "SPECK")
  {
    uint32_t key_schedule[SPECK_ROUNDS_128];
    uint32_t key_words[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
    speckExpand(key_words, key_schedule, SPECK_ROUNDS_128);
    uint32_t block[2];
    memcpy(block, data, 8);
    speckEncryptDecrypt(block, key_schedule, SPECK_ROUNDS_128);
    memcpy(data, block, 8);
  }
  else if (method == "SIMON")
  {
    uint32_t key_schedule[SIMON_ROUNDS_128];
    uint32_t key_words[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
    simonExpand(key_words, key_schedule, SIMON_ROUNDS_128);
    uint32_t block[2];
    memcpy(block, data, 8);
    simonEncryptDecrypt(block, key_schedule, SIMON_ROUNDS_128);
    memcpy(data, block, 8);
  }
}

// Function to generate a session ID
```

```
String generateSessionID()
{
  String sessionID = "";
  for (int i = 0; i < 16; i++)
  {
    sessionID += String(random(0, 16), HEX);
  }
  return sessionID;
}

// Function to check if a session ID is approved
bool isSessionApproved(String sessionID, String clientUUID)
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].sessionID == sessionID && sessions[i].clientUUID ==
clientUUID && sessions[i].approved)
    {
      return true;
    }
  }
  return false;
}

// Function to check if a UUID is approved
bool isUUIDApproved(String uuid)
{
  for (int i = 0; i < approvedUUIDCount; i++)
  {
    if (approvedUUIDs[i] == uuid)
    {
      return true;
    }
  }
  return false;
}

// Function to update the server's SSID and password based on the current
ledger entry
void updateServerCredentials()
{
  WiFi.softAPdisconnect(true);
  WiFi.softAP(ledger[currentLedgerIndex].ssid.c_str(),
ledger[currentLedgerIndex].password.c_str());
  Serial.println("Server SSID and Password updated to:");
  Serial.println("SSID: " + ledger[currentLedgerIndex].ssid);
  Serial.println("Password: " + ledger[currentLedgerIndex].password);
}
```

```
// Function to start the honeypot SSID
void startHoneypot()
{
  WiFi.softAPdisconnect(true);
  WiFi.softAP("Honeypot_SSID", "Honeypot_Password");
  Serial.println("Honeypot SSID and Password activated:");
  Serial.println("Honeypot SSID: Honeypot_SSID");
  Serial.println("Honeypot Password: Honeypot_Password");
}

// Function to stop the honeypot SSID
void stopHoneypot()
{
  WiFi.softAPdisconnect(true);
  updateServerCredentials();
  Serial.println("Honeypot SSID deactivated. Server SSID and Password
restored.");
}

// Function to notify all approved clients of a ledger update
void notifyApprovedClients()
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].approved && sessions[i].clientConnection.connected())
    {
      sessions[i].clientConnection.println("Ledger Update");
      sessions[i].clientConnection.println(currentLedgerIndex);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].ssid);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].password
);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].encrypti
onKey);
      sessions[i].clientConnection.println(ledger[currentLedgerIndex].encrypti
onMethod);
    }
  }
}

// Function to handle compromised credentials and trigger process
void handleCompromisedCredentials(String sessionID, String clientUUID)
{
  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].sessionID == sessionID && sessions[i].clientUUID ==
clientUUID)
    {
      Serial.println("Session ID " + sessionID + " marked as compromised.");
      sessions[i].approved = false;
      currentLedgerIndex = (currentLedgerIndex + 1) % 5;
      updateServerCredentials();
      notifyApprovedClients();
      startHoneypot();
      break;
    }
  }
  for (int i = 0; i < sessionCount; i++)
  {
    sessions[i].reconnected = false;
  }
  Serial.println("All sessions marked as needing reconnection.");
}

// Function to check for desynchronization and send error code
void checkDesynchronization(WiFiClient &client, String clientUUID)
{
  for (int i = 0; i < approvedUUIDCount; i++)
  {
    if (approvedUUIDs[i] == clientUUID && !uuidConnected[i])
    {
      String clientSSID = client.readStringUntil('\n');
      String clientPassword = client.readStringUntil('\n');

      if (clientSSID != ledger[currentLedgerIndex].ssid || clientPassword !=
ledger[currentLedgerIndex].password)
      {
        Serial.println("Desync detected for UUID: " + clientUUID);
        client.println("Error: Desync detected. Please reconnect.");
        client.stop();
        return;
      }
      uuidConnected[i] = true;
      Serial.println("UUID " + clientUUID + " connected successfully.");
    }
  }
}

// Function to handle new incoming connections
void handleNewConnection(WiFiClient client)
{
  String sessionID = generateSessionID();
  client.println(sessionID);

  String clientInfo = client.readStringUntil('\n');
  String clientUUID = client.readStringUntil('\n');
  String receivedSessionID = client.readStringUntil('\n');

  String encryptionMethod = ledger[currentLedgerIndex].encryptionMethod;
  client.println(encryptionMethod);

  if (isUUIDApproved(clientUUID))
  {
    if (!isSessionApproved(receivedSessionID, clientUUID))
    {
      sessions[sessionCount].sessionID = sessionID;
      sessions[sessionCount].clientUUID = clientUUID;
      sessions[sessionCount].approved = true;
      sessions[sessionCount].clientConnection = client;
      sessions[sessionCount].reconnected = false;
      sessions[sessionCount].sessionExpirationTime = millis() + 60000;
      sessionCount++;

      Serial.println("Session ID " + sessionID + " for UUID " + clientUUID + " 
connected.");

      checkDesynchronization(client, clientUUID);

      client.println("Connection Successful");

      client.println(ledger[currentLedgerIndex].ssid);
      client.println(ledger[currentLedgerIndex].password);
      client.println(ledger[currentLedgerIndex].encryptionKey);
      client.println(ledger[currentLedgerIndex].encryptionMethod);

      for (int i = 0; i < sessionCount; i++)
      {
        if (sessions[i].clientConnection == client)
        {
          sessions[i].reconnected = true;
          Serial.println("Session ID " + sessionID + " marked as 
reconnected.");
          break;
        }
      }
    }
    else
    {
      Serial.println("Session ID " + receivedSessionID + " is already 
approved. Skipping reinitialization.");
      client.println("Connection Successful");

      client.println(ledger[currentLedgerIndex].ssid);
      client.println(ledger[currentLedgerIndex].password);
      client.println(ledger[currentLedgerIndex].encryptionKey);
      client.println(ledger[currentLedgerIndex].encryptionMethod);
    }
  }
  else
  {
    Serial.println("Failed connection attempt with Session ID " + sessionID +
" and UUID " + clientUUID);
    client.println("Session ID blocked or compromised");
    client.stop();
  }

  String sensorData = client.readStringUntil('\n');
  encryptDecryptData(&sensorData[0],
ledger[currentLedgerIndex].encryptionMethod,
ledger[currentLedgerIndex].encryptionKey);
  Serial.println("Received Decrypted Sensor Data: " + sensorData);
  client.println("Sensor data received and decrypted successfully.");

  bool allReconnected = true;
  for (int i = 0; i < approvedUUIDCount; i++)
  {
    if (!uuidConnected[i])
    {
      allReconnected = false;
      Serial.println("UUID " + approvedUUIDs[i] + " has not reconnected
yet.");
      break;
    }
  }

  if (allReconnected)
  {
    Serial.println("All approved UUIDs have reconnected.");
    stopHoneypot();
  }
}

void setup()
{
  Serial.begin(115200);
  WiFi.softAP(ledger[currentLedgerIndex].ssid.c_str(),
ledger[currentLedgerIndex].password.c_str());
  server.begin();

  Serial.println("Server started with initial SSID and Password:");
  Serial.println("SSID: " + ledger[currentLedgerIndex].ssid);
  Serial.println("Password: " + ledger[currentLedgerIndex].password);
```

```
}

void loop()
{
  WiFiClient client = server.available();
  if (client)
  {
    handleNewConnection(client);
  }

  for (int i = 0; i < sessionCount; i++)
  {
    if (sessions[i].approved && millis() > sessions[i].sessionExpirationTime
&& !sessions[i].isSleeping)
    {
      Serial.println("Session ID " + sessions[i].sessionID + " expired due to
inactivity.");
      handleCompromisedCredentials(sessions[i].sessionID,
sessions[i].clientUUID);
    }
  }

  delay(2000);
}
```

16.2 Client

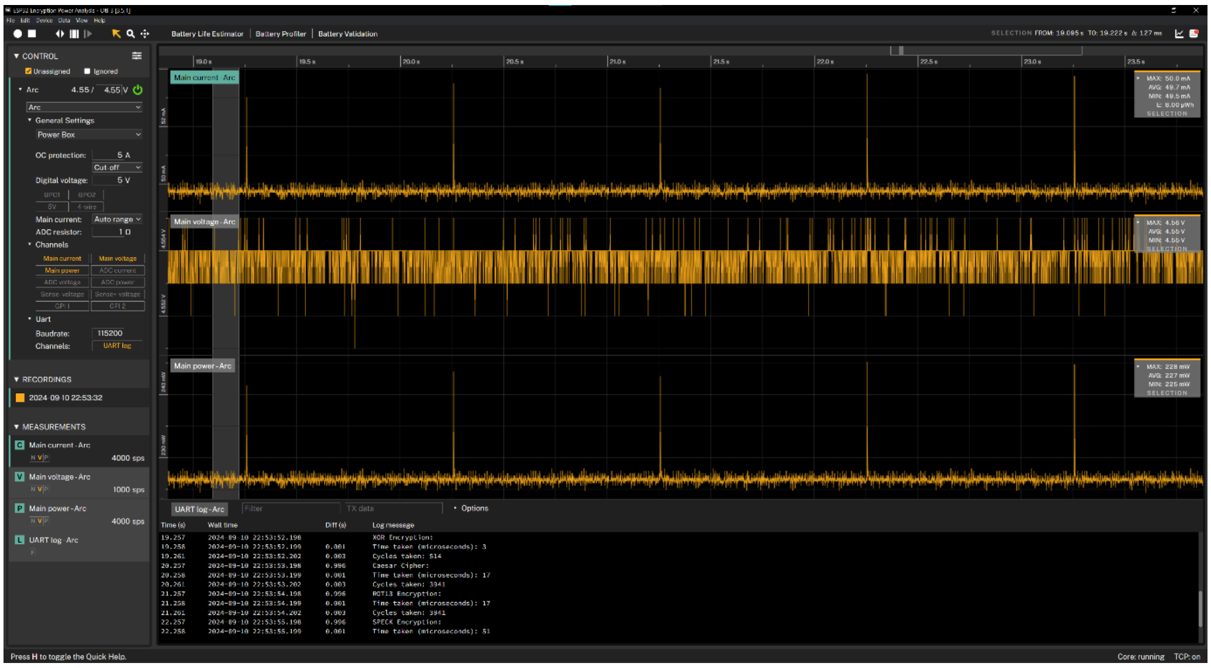
```
#include <WiFi.h>
#include <EEPROM.h>

#define ADC_PIN A0
#define EEPROM_SIZE 512

#define MAX_DATA_POINTS 100

// Server details
const char *host = "192.168.4.1";
const uint16_t port = 80;

// Constants for encryption
#define SPECK_ROUNDS_128 22
#define SIMON_ROUNDS_128 44
#define SPECK_KEY_WORDS 4
#define SIMON_KEY_WORDS 4

WiFiClient client;

// Ledger entries to store received data
struct LedgerEntry
{
  String ssid;
  String password;
  String encryptionKey;
  String encryptionMethod;
};

LedgerEntry storedLedger[5];
int storedLedgerCount = 0;

// Battery Data Structure
struct BatteryData
{
  uint32_t voltage;
  float percentage;
};

BatteryData batteryData[MAX_DATA_POINTS];
int dataPointsCount = 0;

uint32_t getBatteryVoltage()
{
  int adcValue = analogRead(ADC_PIN);
  uint32_t voltage = map(adcValue, 0, 4095, 0, 5000);
  return voltage;
}

float initialMapBatteryCurve(uint32_t voltage)
{
  if (voltage > 4200)
    return 100.0;
  else if (voltage > 4000)
    return 75.0 + (voltage - 4000) * 0.25;
  else if (voltage > 3800)
    return 50.0 + (voltage - 3800) * 0.125;
  else if (voltage > 3600)
    return 25.0 + (voltage - 3600) * 0.125;
  else if (voltage > 3400)
    return (voltage - 3400) * 0.125;
  else
    return 0.0;
}

float nonLinearInterpolation(uint32_t voltage)
{
  float voltageNormalized = (float)(voltage - 3400) / (4200 - 3400);
  return pow(voltageNormalized, 2) * 100.0;
}

void storeBatteryData(uint32_t voltage, float percentage)
{
  if (dataPointsCount < MAX_DATA_POINTS)
  {
    batteryData[dataPointsCount].voltage = voltage;
    batteryData[dataPointsCount].percentage = percentage;
    dataPointsCount++;
  }
  else
  {
    for (int i = 1; i < MAX_DATA_POINTS; i++)
    {
      batteryData[i - 1] = batteryData[i];
    }
    batteryData[MAX_DATA_POINTS - 1].voltage = voltage;
    batteryData[MAX_DATA_POINTS - 1].percentage = percentage;
  }

  EEPROM.put(0, batteryData);
  EEPROM.put(sizeof(batteryData), dataPointsCount);
  EEPROM.commit();
}

float estimateBatteryHealth(uint32_t voltage)
```

```
{
  if (voltage > 4200)
    return 1.0;
  else if (voltage > 4000)
    return 0.75;
  else if (voltage > 3800)
    return 0.5;
  else if (voltage > 3600)
    return 0.25;
  else
    return 0.1;
}

void loadBatteryData()
{
  EEPROM.get(0, batteryData);
  EEPROM.get(sizeof(batteryData), dataPointsCount);
}

float predictBatteryPercentage(uint32_t voltage)
{
  if (dataPointsCount < 2)
  {
    return initialMapBatteryCurve(voltage);
  }

  return nonLinearInterpolation(voltage);
}

void enterAdaptiveSleepMode(float batteryPercentage, float batteryHealth)
{
  uint64_t sleepDuration;

  if (batteryPercentage > 75.0)
  {
    Serial.println("Battery sufficient, normal operation.");
    sleepDuration = 10000000;
  }
  else if (batteryPercentage > 50.0)
  {
    Serial.println("Entering light sleep mode.");
    sleepDuration = batteryHealth * 20000000;
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_light_sleep_start();
  }
  else if (batteryPercentage > 25.0)
  {
    Serial.println("Entering deep sleep mode.");
    sleepDuration = batteryHealth * 40000000;
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_deep_sleep_start();
  }
  else
  {
    Serial.println("Entering ULP mode, maximizing battery life.");
    sleepDuration = batteryHealth * 60000000;
    esp_sleep_enable_timer_wakeup(sleepDuration);
    esp_deep_sleep_start();
  }
}

void sendLowBatteryAlert()
{
  if (client.connected())
  {
    client.println("Low Battery Alert");
  }
}

// XOR Encryption/Decryption
void xorEncryptionDecryption(char *data, String key)
{
  for (int i = 0; i < strlen(data); i++)
  {
    data[i] ^= key[0];
  }
}

// Caesar Cipher Encryption/Decryption
void caesarCipherEncryptionDecryption(char *data, int shift)
{
  for (int i = 0; i < strlen(data); i++)
  {
    if (isalpha(data[i]))
    {
      char offset = isupper(data[i]) ? 'A' : 'a';
      data[i] = (data[i] - offset + shift) % 26 + offset;
    }
  }
}

// ROT13 Encryption/Decryption
void rot13EncryptionDecryption(char *data)
{
  caesarCipherEncryptionDecryption(data, 13);
}
```

```
// SPECK Encryption/Decryption
void speckRound(uint32_t &x, uint32_t &y, uint32_t k)
{
  x = (x >> 8) | (x << (32 - 8));
  x += y;
  x ^= k;
  y = (y << 3) | (y >> (32 - 3));
  y ^= x;
}

void speckExpand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SPECK_KEY_WORDS - 1];
  for (i = 0; i < (SPECK_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    speckRound(a[i % (SPECK_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void speckEncryptDecrypt(uint32_t data[2], uint32_t const key_schedule[],
uint8_t rounds)
{
  for (uint8_t i = 0; i < rounds; i++)
  {
    speckRound(data[1], data[0], key_schedule[i]);
  }
}

// SIMON Encryption/Decryption
void simonRound(uint32_t &x, uint32_t &y, uint32_t k)
{
  uint32_t tmp = (x << 1) | (x >> (32 - 1));
  tmp &= (tmp << 8);
  tmp ^= y;
  y ^= k;
  y = (y >> 1) | (y << (32 - 1));
}

void simonExpand(uint32_t const K[], uint32_t S[], uint8_t rounds)
{
  uint32_t i, b = K[0];
  uint32_t a[SIMON_KEY_WORDS - 1];
  for (i = 0; i < (SIMON_KEY_WORDS - 1); i++)
  {
    a[i] = K[i + 1];
  }
  S[0] = b;
  for (i = 0; i < rounds - 1; i++)
  {
    simonRound(a[i % (SIMON_KEY_WORDS - 1)], b, i);
    S[i + 1] = b;
  }
}

void simonEncryptDecrypt(uint32_t data[2], uint32_t const key_schedule[],
uint8_t rounds)
{
  for (uint8_t i = 0; i < rounds; i++)
  {
    simonRound(data[1], data[0], key_schedule[i]);
  }
}

void encryptDecryptData(char *data, const String &method, const String &key)
{
  if (method == "XOR")
  {
    xorEncryptionDecryption(data, key);
  }
  else if (method == "Caesar")
  {
    caesarCipherEncryptionDecryption(data, key.toInt());
  }
  else if (method == "ROT13")
  {
    rot13EncryptionDecryption(data);
  }
  else if (method == "SPECK")
  {
    uint32_t key_schedule[SPECK_ROUNDS_128];
    uint32_t key_words[SPECK_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
    speckExpand(key_words, key_schedule, SPECK_ROUNDS_128);
    uint32_t block[2];
    memcpy(block, data, 8);
    speckEncryptDecrypt(block, key_schedule, SPECK_ROUNDS_128);
    memcpy(data, block, 8);
  }
  else if (method == "SIMON")
  {
    uint32_t key_schedule[SIMON_ROUNDS_128];
    uint32_t key_words[SIMON_KEY_WORDS] = {0x01020304, 0x05060708, 0x090A0B0C,
0x0D0E0F10};
    simonExpand(key_words, key_schedule, SIMON_ROUNDS_128);
    uint32_t block[2];
    memcpy(block, data, 8);
    simonEncryptDecrypt(block, key_schedule, SIMON_ROUNDS_128);
    memcpy(data, block, 8);
  }
}

bool connectToWiFi(const char *ssid, const char *password)
{
  WiFi.begin(ssid, password);

  int attempts = 0;
  while (WiFi.status() != WL_CONNECTED && attempts < 20)
  {
    delay(500);
    Serial.print(".");
    attempts++;
  }

  if (WiFi.status() == WL_CONNECTED)
  {
    Serial.println("\nConnected to Wi-Fi");
    return true;
  }

  Serial.println("\nFailed to connect to Wi-Fi");
  return false;
}

bool connectToWiFiWithStoredLedger()
{
  if (storedLedgerCount > 0)
  {
    return connectToWiFi(storedLedger[storedLedgerCount - 1].ssid.c_str(),
storedLedger[storedLedgerCount - 1].password.c_str());
  }
  return false;
}

void handleDesync()
{
  Serial.println("Handling desync, attempting to connect to honeypot...");
  if (connectToWiFi("Honeypot_SSID", "Honeypot_Password")) // Use the actual
honeypot SSID and Password
  {
    if (client.connect(host, port))
    {
      Serial.println("Connected to server via honeypot");

      String uuid = "UUID1-1234-5678-91011";
      client.println(uuid);

      for (int i = 0; i < 5; i++)
      {
        storedLedger[i].ssid = client.readStringUntil('\n');
        storedLedger[i].password = client.readStringUntil('\n');
        storedLedger[i].encryptionKey = client.readStringUntil('\n');
        storedLedger[i].encryptionMethod = client.readStringUntil('\n');
        storedLedgerCount++;
      }

      Serial.println("Updated ledger received:");
      for (int i = 0; i < storedLedgerCount; i++)
      {
        Serial.println("SSID: " + storedLedger[i].ssid);
        Serial.println("Password: " + storedLedger[i].password);
        Serial.println("Encryption Key: " + storedLedger[i].encryptionKey);
        Serial.println("Encryption Method: " +
storedLedger[i].encryptionMethod);
      }

      if (connectToWiFiWithStoredLedger())
      {
        Serial.println("Reconnected to server using updated ledger
credentials");
      }
      else
      {
        Serial.println("Failed to reconnect to server after receiving updated
ledger");
      }
    }
    else
    {
      Serial.println("Failed to connect to server via honeypot");
    }
  }
  else
  {
    Serial.println("Failed to connect to honeypot SSID");
  }
}

void handleLedgerUpdate()
{
  if (client.connected())
  {
    String ledgerUpdate = client.readStringUntil('\n');
    if (ledgerUpdate == "Ledger Update")
    {
      int newIndex = client.readStringUntil('\n').toInt();
      storedLedger[newIndex].ssid = client.readStringUntil('\n');
      storedLedger[newIndex].password = client.readStringUntil('\n');
      storedLedger[newIndex].encryptionKey = client.readStringUntil('\n');
      storedLedger[newIndex].encryptionMethod = client.readStringUntil('\n');

      Serial.println("Ledger updated:");
      Serial.println("New SSID: " + storedLedger[newIndex].ssid);
      Serial.println("New Password: " + storedLedger[newIndex].password);
      Serial.println("New Encryption Key: " +
storedLedger[newIndex].encryptionKey);
      Serial.println("New Encryption Method: " +
storedLedger[newIndex].encryptionMethod);

      if (!connectToWiFiWithStoredLedger())
      {
        Serial.println("Error: Desynced from server. Unable to reconnect.");
        handleDesync();
      }
    }
  }
}

void setup()
{
  Serial.begin(115200);
  EEPROM.begin(EEPROM_SIZE);

  loadBatteryData();

  Serial.println("Starting battery management system...");

  if (!connectToWiFiWithStoredLedger())
  {
    Serial.println("Error: Initial connection failed. Desynced from server.");
    handleDesync();
    return;
  }

  if (client.connect(host, port))
  {
    Serial.println("Connected to server");

    String sessionID = client.readStringUntil('\n');
    Serial.println("Received Session ID: " + sessionID);

    String processingPower = "HIGH";
    String uuid = "UUID1-1234-5678-91011";
    client.println(processingPower);
    client.println(uuid);
    client.println(sessionID);

    String encryptionMethod = client.readStringUntil('\n');
    Serial.println("Encryption Method: " + encryptionMethod);

    String response = client.readStringUntil('\n');
    Serial.println("Server Response: " + response);

    if (response == "Connection Successful")
    {
      for (int i = 0; i < 5; i++)
      {
        storedLedger[i].ssid = client.readStringUntil('\n');
        storedLedger[i].password = client.readStringUntil('\n');
        storedLedger[i].encryptionKey = client.readStringUntil('\n');
        storedLedger[i].encryptionMethod = client.readStringUntil('\n');
        storedLedgerCount++;
      }

      Serial.println("Received full ledger:");
      for (int i = 0; i < storedLedgerCount; i++)
      {
        Serial.println("SSID: " + storedLedger[i].ssid);
        Serial.println("Password: " + storedLedger[i].password);
        Serial.println("Encryption Key: " + storedLedger[i].encryptionKey);
        Serial.println("Encryption Method: " +
storedLedger[i].encryptionMethod);
      }

      uint32_t voltage = getBatteryVoltage();
      float batteryPercentage = predictBatteryPercentage(voltage);
      float batteryHealth = estimateBatteryHealth(voltage);

      // Dynamic Encryption Mode Selection
      if (batteryPercentage < 15.0)
      {
        storedLedger[0].encryptionMethod = "XOR"; // Ultra-lightweight
      }
      else if (batteryPercentage < 30.0)
      {
        storedLedger[0].encryptionMethod = "Caesar"; // Simple shift
      }
      else if (batteryPercentage < 50.0)
      {
        storedLedger[0].encryptionMethod = "ROT13"; // Moderate cost
      }
      else if (batteryPercentage < 75.0)
      {
        storedLedger[0].encryptionMethod = "SPECK"; // Stronger
      }
      else
      {
        storedLedger[0].encryptionMethod = "SIMON"; // Most robust
      }

      Serial.println("Selected Encryption Method: " +
storedLedger[0].encryptionMethod);

      client.println("EncryptionMode: " + storedLedger[0].encryptionMethod);

      if (batteryPercentage < 10.0)
      {
        sendLowBatteryAlert();
      }

      storeBatteryData(voltage, batteryPercentage);
      enterAdaptiveSleepMode(batteryPercentage, batteryHealth);

      // Sensor data to be encrypted and sent
      char sensorData[] = "The quick brown fox jumps over the lazy dog
1234567890!@#$%^&*()_+-=[]{}|;':,.<>/?\n\t";
      encryptDecryptData(sensorData, encryptionMethod,
storedLedger[0].encryptionKey);

      client.println(sensorData);

      String confirmation = client.readStringUntil('\n');
      Serial.println("Server Confirmation: " + confirmation);
    }
  }
  else
  {
    Serial.println("Failed to connect to server");
  }
}

void loop()
{
  if (client.connected())
  {
    handleLedgerUpdate();
  }
  else
  {
    Serial.println("Disconnected from server");

    if (connectToWiFiWithStoredLedger() && client.connect(host, port))
    {
      Serial.println("Reconnected to server using ledger credentials");
    }
    else
    {
      Serial.println("Failed to reconnect to server");
      handleDesync();
    }
  }

  delay(5000);
}
```

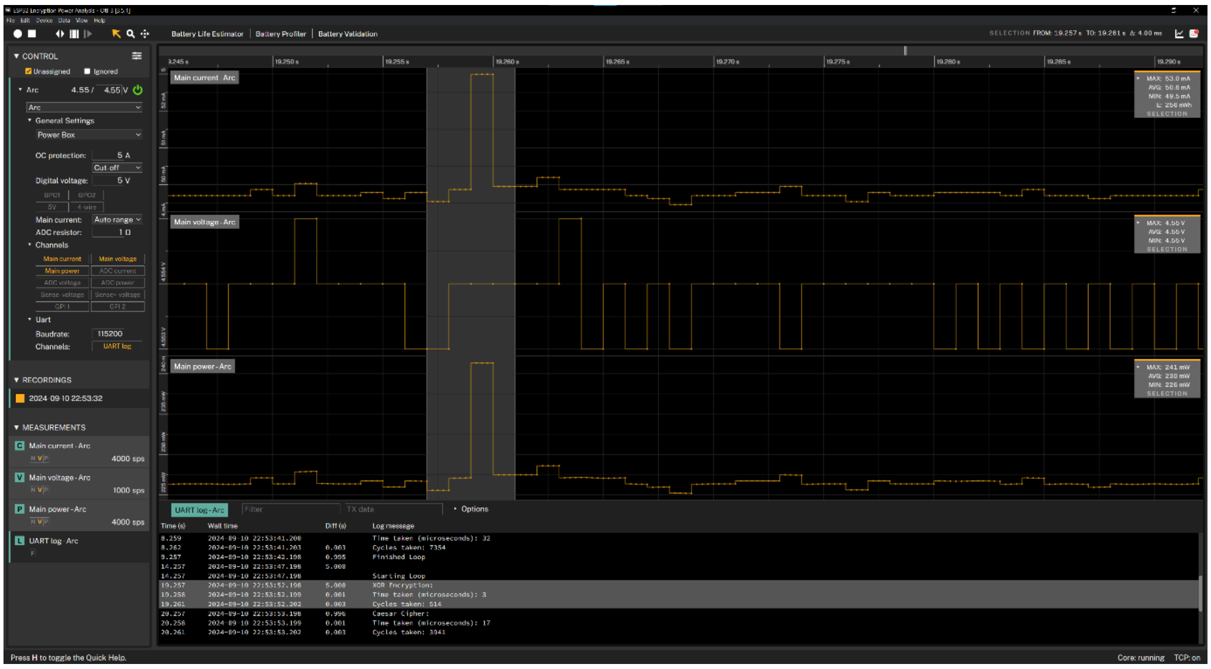
Appendix 17: Encryption Power Performance Summary

17.1 ESP32 Encryption Power Performance



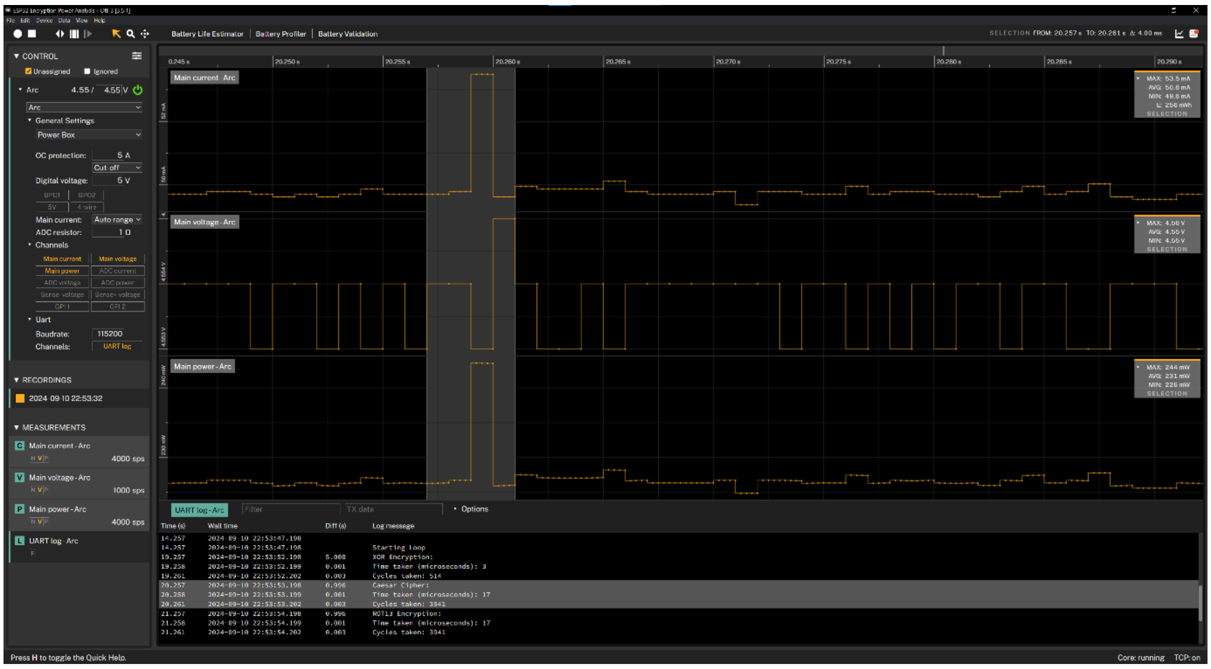
```
Time (s)   Wall time                 Diff (s)   Log message
19.257     2024-09-10 22:53:52.198              XOR Encryption:
19.258     2024-09-10 22:53:52.199   0.001      Time taken (microseconds): 3
19.261     2024-09-10 22:53:52.202   0.003      Cycles taken: 514
20.257     2024-09-10 22:53:53.198   0.996      Caesar Cipher:
20.258     2024-09-10 22:53:53.199   0.001      Time taken (microseconds): 17
20.261     2024-09-10 22:53:53.202   0.003      Cycles taken: 3941
21.257     2024-09-10 22:53:54.198   0.996      ROT13 Encryption:
21.258     2024-09-10 22:53:54.199   0.001      Time taken (microseconds): 17
21.261     2024-09-10 22:53:54.202   0.003      Cycles taken: 3941
22.257     2024-09-10 22:53:55.198   0.996      SPECK Encryption:
22.258     2024-09-10 22:53:55.199   0.001      Time taken (microseconds): 51
```

Figure Appendix 17.1: Idel ESP32 Encryption Power Performance



```
Time (s)   Wall time                 Diff (s)   Log message
8.259      2024-09-10 22:53:41.200              Time taken (microseconds): 32
8.262      2024-09-10 22:53:41.203   0.003      Cycles taken: 7354
9.257      2024-09-10 22:53:42.198   0.995      Finished Loop
14.257     2024-09-10 22:53:47.198   5.000
14.257     2024-09-10 22:53:47.198              Starting Loop
19.257     2024-09-10 22:53:52.198   5.000      XOR Encryption:
19.258     2024-09-10 22:53:52.199   0.001      Time taken (microseconds): 3
19.261     2024-09-10 22:53:52.202   0.003      Cycles taken: 514
20.257     2024-09-10 22:53:53.198   0.996      Caesar Cipher:
20.258     2024-09-10 22:53:53.199   0.001      Time taken (microseconds): 17
20.261     2024-09-10 22:53:53.202   0.003      Cycles taken: 3941
```

Figure Appendix 17.2: XOR ESP32 Encryption Power Performance



SELECTION FROM: 20.257 s TO: 20.261 s Δ: 4.00 ms

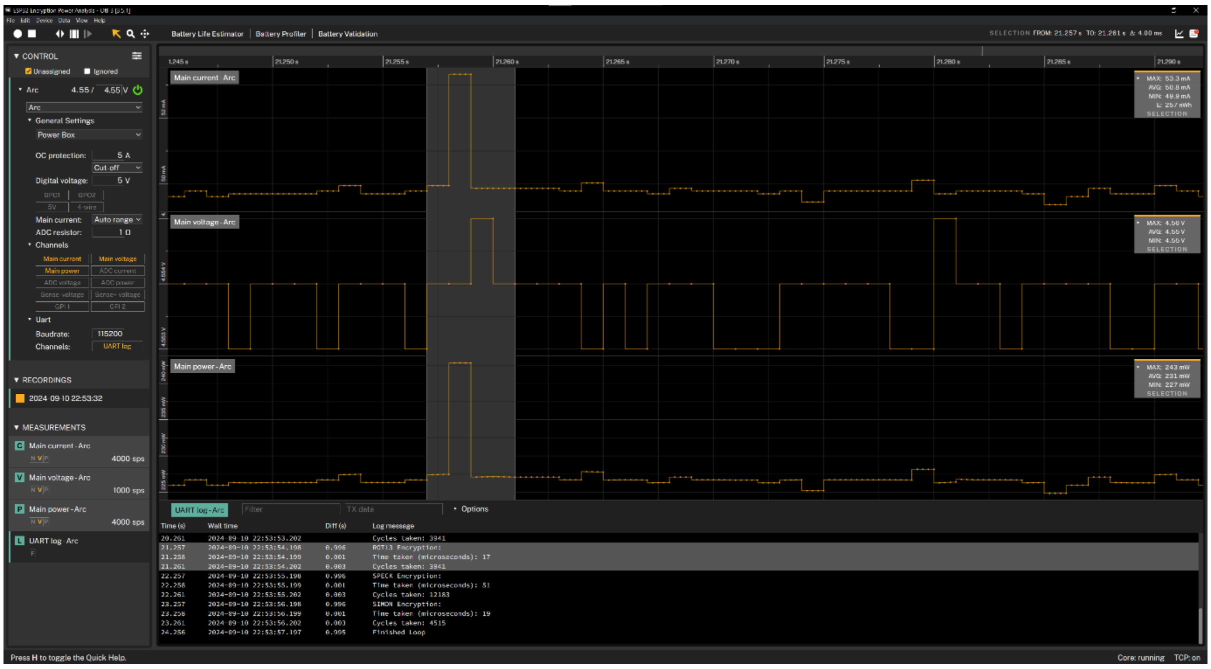
MAX: 53.5 mA
AVG: 50.8 mA
MIN: 49.8 mA
E: 256 nWh

MAX: 4.56 V
AVG: 4.55 V
MIN: 4.55 V

MAX: 244 mW
AVG: 231 mW
MIN: 226 mW

```
Time (s)  Wall time                 Diff (s)  Log message
14.257    2024-09-10 22:53:47.198
14.257    2024-09-10 22:53:47.198             Starting Loop
19.257    2024-09-10 22:53:52.198   5.000     XOR Encryption:
19.258    2024-09-10 22:53:52.199   0.001     Time taken (microseconds): 3
19.261    2024-09-10 22:53:52.202   0.003     Cycles taken: 514
20.257    2024-09-10 22:53:53.198   0.996     Caesar Cipher:
20.258    2024-09-10 22:53:53.199   0.001     Time taken (microseconds): 17
20.261    2024-09-10 22:53:53.202   0.003     Cycles taken: 3941
21.257    2024-09-10 22:53:54.198   0.996     ROT13 Encryption:
21.258    2024-09-10 22:53:54.199   0.001     Time taken (microseconds): 17
21.261    2024-09-10 22:53:54.202   0.003     Cycles taken: 3941
```

Figure Appendix 17.3: Caesar ESP32 Encryption Power Performance



SELECTION FROM: 21.257 s TO: 21.261 s Δ: 4.00 ms

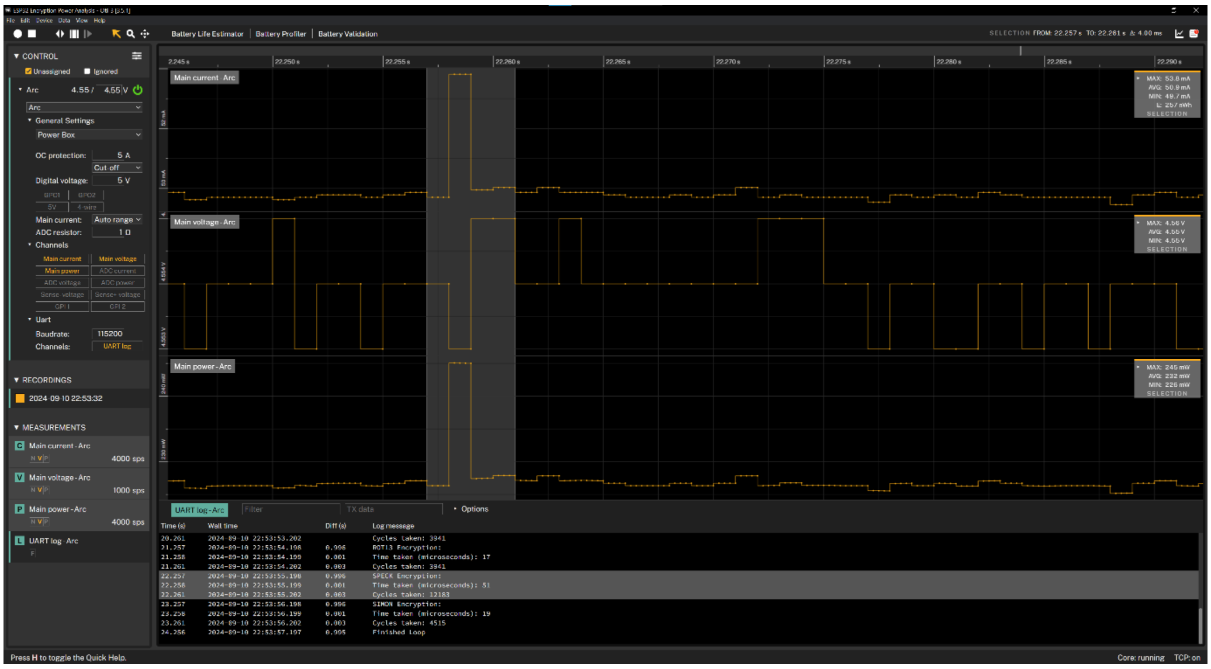
MAX: 53.3 mA
AVG: 50.8 mA
MIN: 49.9 mA
E: 257 nWh

MAX: 4.56 V
AVG: 4.55 V
MIN: 4.55 V

MAX: 243 mW
AVG: 231 mW
MIN: 227 mW

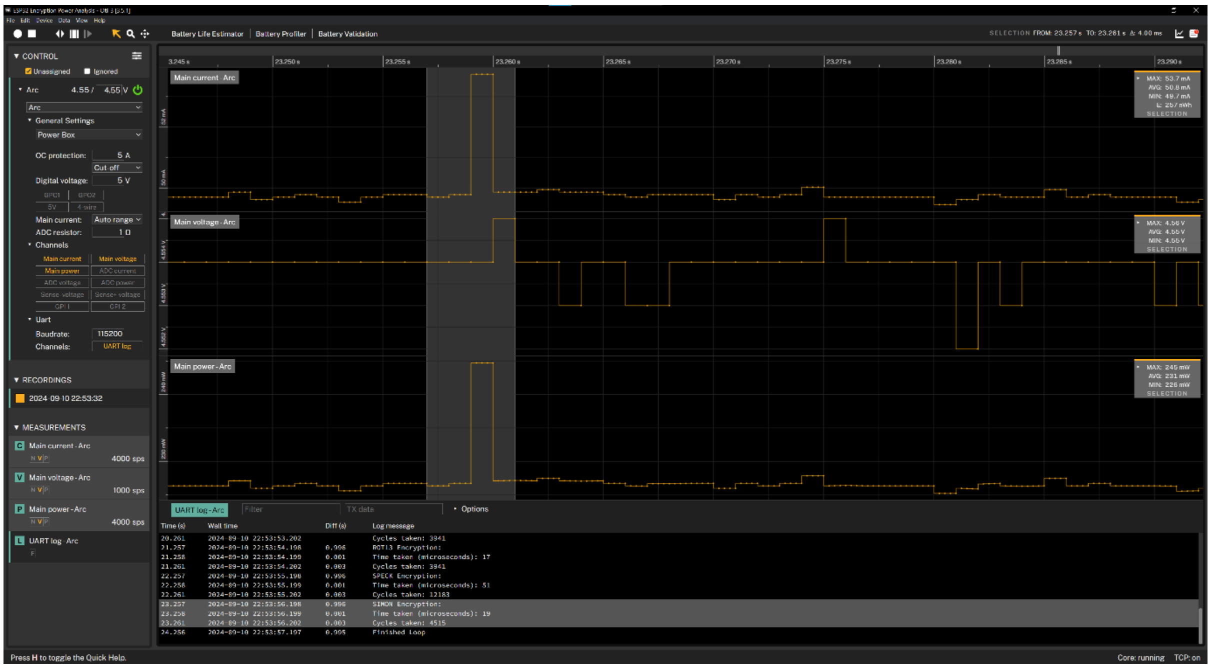
```
Time (s)  Wall time                 Diff (s)  Log message
20.261    2024-09-10 22:53:53.202             Cycles taken: 3941
21.257    2024-09-10 22:53:54.198   0.996     ROT13 Encryption:
21.258    2024-09-10 22:53:54.199   0.001     Time taken (microseconds): 17
21.261    2024-09-10 22:53:54.202   0.003     Cycles taken: 3941
22.257    2024-09-10 22:53:55.198   0.996     SPECK Encryption:
22.258    2024-09-10 22:53:55.199   0.001     Time taken (microseconds): 51
22.261    2024-09-10 22:53:55.202   0.003     Cycles taken: 12183
23.257    2024-09-10 22:53:56.198   0.996     SIMON Encryption:
23.258    2024-09-10 22:53:56.199   0.001     Time taken (microseconds): 19
23.261    2024-09-10 22:53:56.202   0.003     Cycles taken: 4515
24.256    2024-09-10 22:53:57.197   0.995     Finished Loop
```

Figure Appendix 17.4: ROT13 ESP32 Encryption Power Performance



```
Time (s)   Wall time                  Diff (s)   Log message
20.261     2024-09-10 22:53:53.202               Cycles taken: 3941
21.257     2024-09-10 22:53:54.198    0.996      ROT13 Encryption:
21.258     2024-09-10 22:53:54.199    0.001      Time taken (microseconds): 17
21.261     2024-09-10 22:53:54.202    0.003      Cycles taken: 3941
22.257     2024-09-10 22:53:55.198    0.996      SPECK Encryption:
22.258     2024-09-10 22:53:55.199    0.001      Time taken (microseconds): 51
22.261     2024-09-10 22:53:55.202    0.003      Cycles taken: 12183
23.257     2024-09-10 22:53:56.198    0.996      SIMON Encryption:
23.258     2024-09-10 22:53:56.199    0.001      Time taken (microseconds): 19
23.261     2024-09-10 22:53:56.202    0.003      Cycles taken: 4515
24.256     2024-09-10 22:53:57.197    0.995      Finished Loop
```

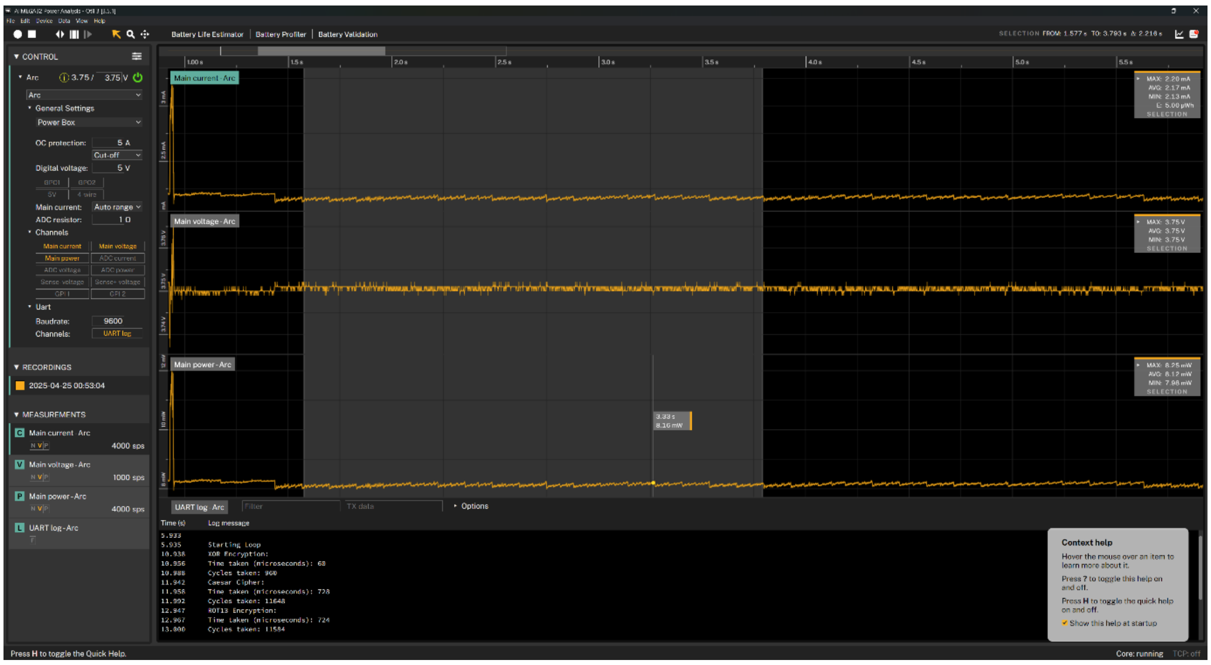
Figure Appendix 17.5: SPECK ESP32 Encryption Power Performance



```
Time (s)   Wall time                  Diff (s)   Log message
20.261     2024-09-10 22:53:53.202               Cycles taken: 3941
21.257     2024-09-10 22:53:54.198    0.996      ROT13 Encryption:
21.258     2024-09-10 22:53:54.199    0.001      Time taken (microseconds): 17
21.261     2024-09-10 22:53:54.202    0.003      Cycles taken: 3941
22.257     2024-09-10 22:53:55.198    0.996      SPECK Encryption:
22.258     2024-09-10 22:53:55.199    0.001      Time taken (microseconds): 51
22.261     2024-09-10 22:53:55.202    0.003      Cycles taken: 12183
23.257     2024-09-10 22:53:56.198    0.996      SIMON Encryption:
23.258     2024-09-10 22:53:56.199    0.001      Time taken (microseconds): 19
23.261     2024-09-10 22:53:56.202    0.003      Cycles taken: 4515
24.256     2024-09-10 22:53:57.197    0.995      Finished Loop
```

Figure Appendix 17.6: SIMON ESP32 Encryption Power Performance

17.2 ATMEGA328 Encryption Power Performance



SELECTION FROM: 1.577 s TO: 3.793 s Δ: 2.216 s

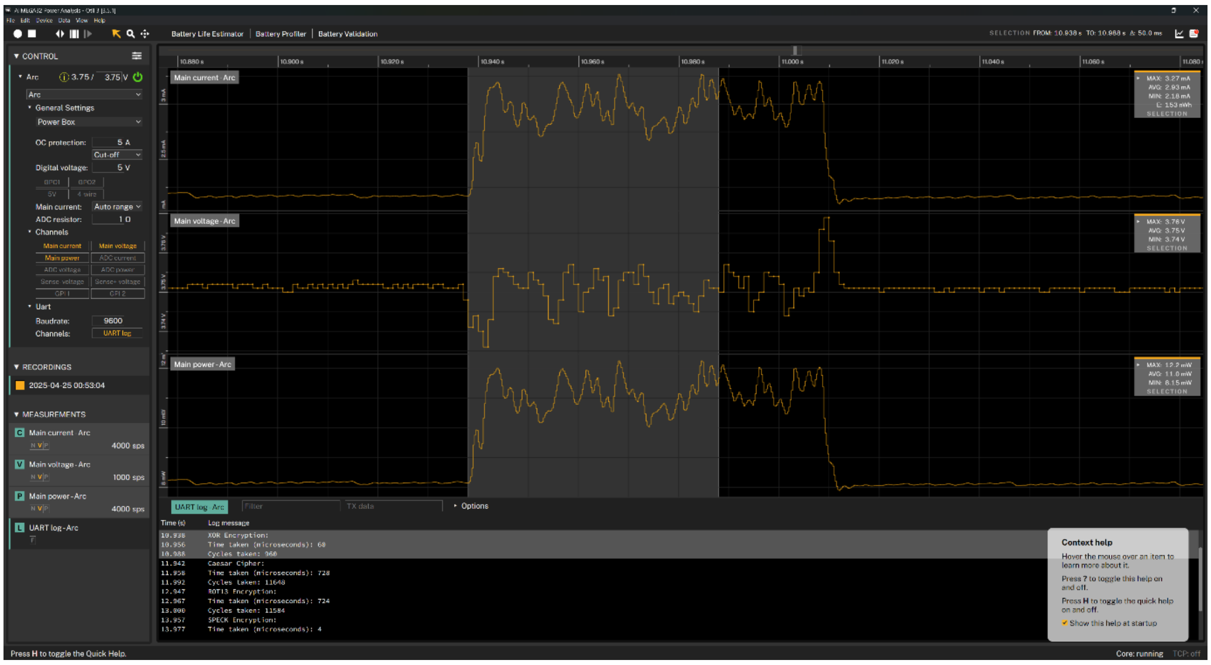
Main current - Arc: MAX: 2.20 mA AVG: 2.17 mA MIN: 2.13 mA E: 5.00 µWh

Main voltage - Arc: MAX: 3.75 V AVG: 3.75 V MIN: 3.75 V

Main power - Arc: MAX: 8.25 mW AVG: 8.12 mW MIN: 7.98 mW

```
Time (s)    Log message
5.933
5.935       Starting Loop
10.938      XOR Encryption:
10.956      Time taken (microseconds): 68
10.988      Cycles taken: 960
11.942      Caesar Cipher:
11.958      Time taken (microseconds): 728
11.992      Cycles taken: 11648
12.947      ROT13 Encryption:
12.967      Time taken (microseconds): 724
13.000      Cycles taken: 11584
```

Figure Appendix 17.7: Idel ATMEGA328 Encryption Power Performance



SELECTION FROM: 10.938 s TO: 10.988 s Δ: 50.0 ms

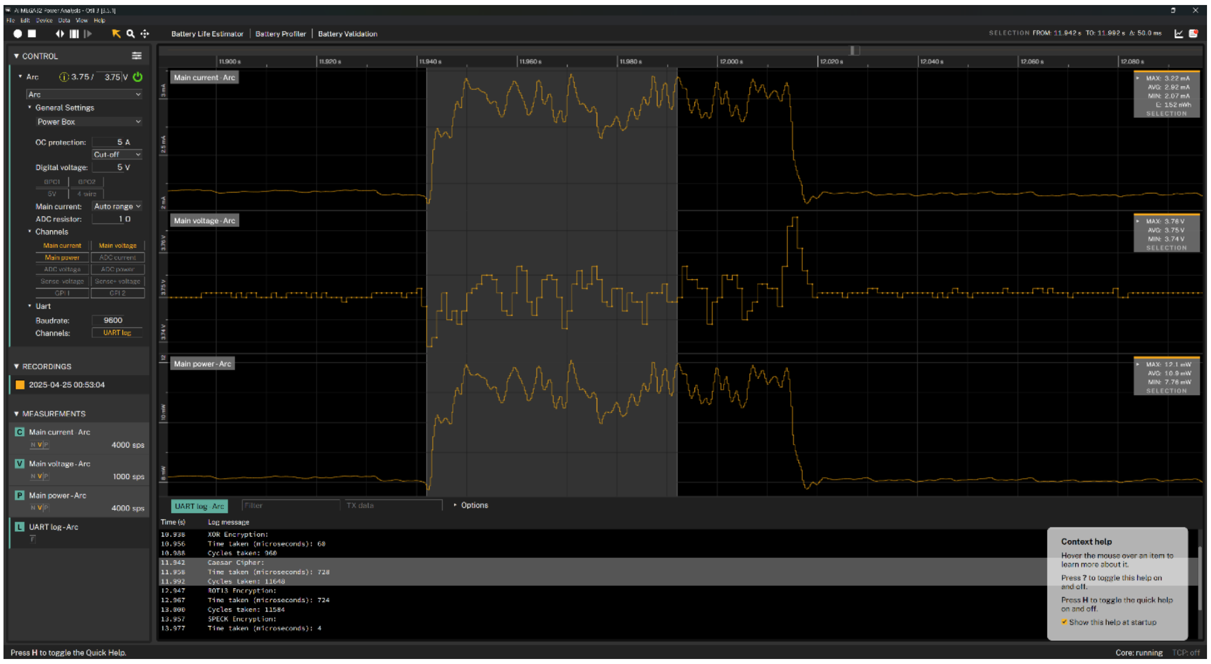
Main current - Arc: MAX: 3.27 mA AVG: 2.93 mA MIN: 2.18 mA E: 153 nWh

Main voltage - Arc: MAX: 3.76 V AVG: 3.75 V MIN: 3.74 V

Main power - Arc: MAX: 12.2 mW AVG: 11.0 mW MIN: 8.15 mW

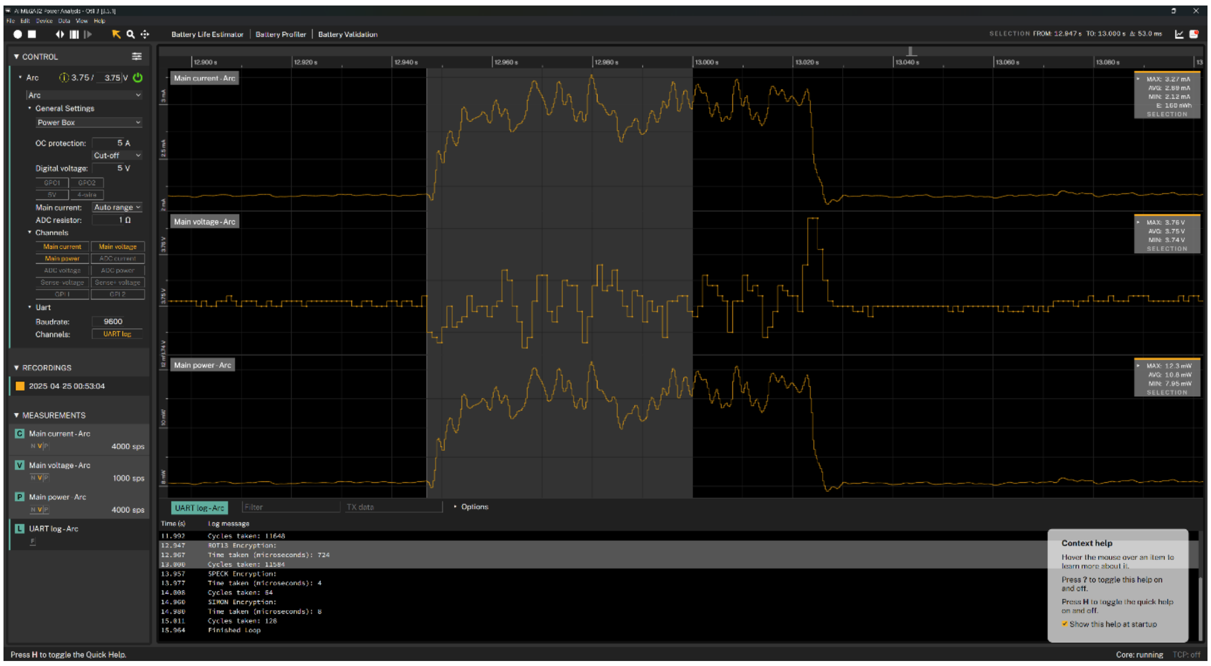
```
Time (s)    Log message
10.938      XOR Encryption:
10.956      Time taken (microseconds): 68
10.988      Cycles taken: 960
11.942      Caesar Cipher:
11.958      Time taken (microseconds): 728
11.992      Cycles taken: 11648
12.947      ROT13 Encryption:
12.967      Time taken (microseconds): 724
13.000      Cycles taken: 11584
13.957      SPECK Encryption:
13.977      Time taken (microseconds): 4
```

Figure Appendix 17.8: XOR ATMEGA328 Encryption Power Performance



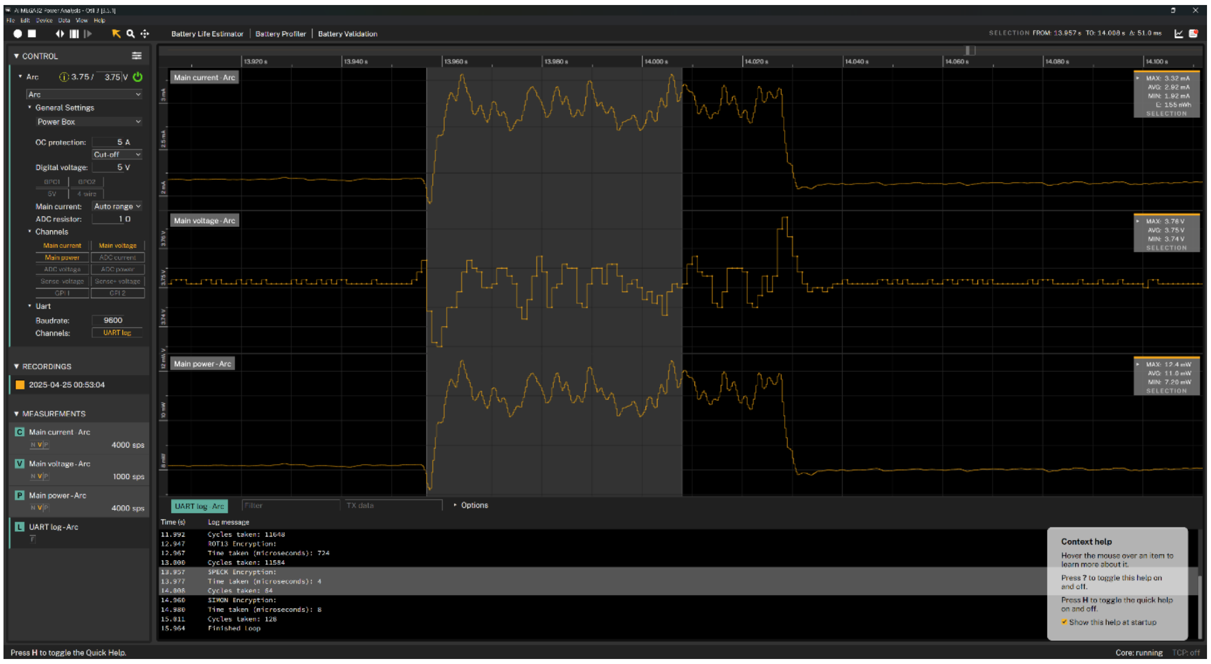
```
Time (s)    Log message
10.938      XOR Encryption:
10.956      Time taken (microseconds): 60
10.988      Cycles taken: 960
11.942      Caesar Cipher:
11.958      Time taken (microseconds): 728
11.992      Cycles taken: 11648
12.947      ROT13 Encryption:
12.967      Time taken (microseconds): 724
13.000      Cycles taken: 11584
13.957      SPECK Encryption:
13.977      Time taken (microseconds): 4
```

Figure Appendix 17.9: Caesar ATMEGA328 Encryption Power Performance



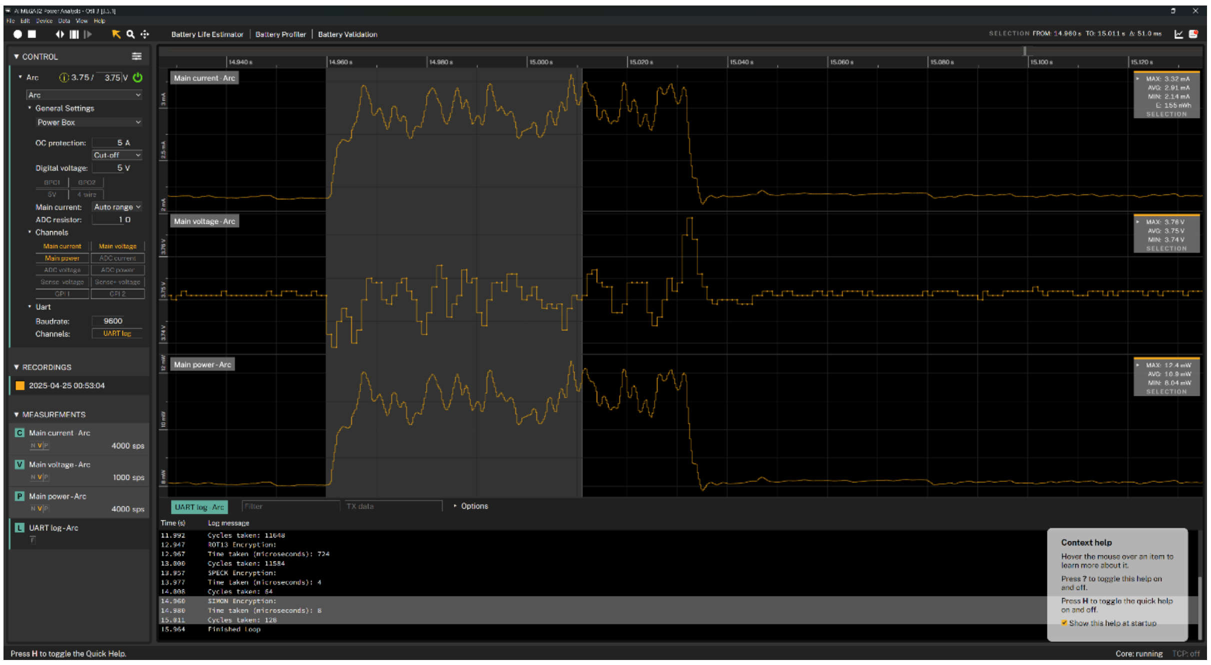
```
Time (s)    Log message
11.992      Cycles taken: 11648
12.947      ROT13 Encryption:
12.967      Time taken (microseconds): 724
13.000      Cycles taken: 11584
13.957      SPECK Encryption:
13.977      Time taken (microseconds): 4
14.008      Cycles taken: 64
14.960      SIMON Encryption:
14.980      Time taken (microseconds): 8
15.011      Cycles taken: 128
15.964      Finished Loop
```

Figure Appendix 17.10: ROT13 ATMEGA328 Encryption Power Performance



```
Time (s)   Log message
11.992     Cycles taken: 11648
12.947     ROT13 Encryption:
12.967     Time taken (microseconds): 724
13.800     Cycles taken: 11584
13.957     SPECK Encryption:
13.977     Time taken (microseconds): 4
14.008     Cycles taken: 64
14.960     SIMON Encryption:
14.980     Time taken (microseconds): 8
15.011     Cycles taken: 128
15.964     Finished Loop
```

Figure Appendix 17.11: SPECK ATMEGA328 Encryption Power Performance



```
Time (s)   Log message
11.992     Cycles taken: 11648
12.947     ROT13 Encryption:
12.967     Time taken (microseconds): 724
13.800     Cycles taken: 11584
13.957     SPECK Encryption:
13.977     Time taken (microseconds): 4
14.008     Cycles taken: 64
14.960     SIMON Encryption:
14.980     Time taken (microseconds): 8
15.011     Cycles taken: 128
15.964     Finished Loop
```

Figure Appendix 17.12: SPECK ATMEGA328 Encryption Power Performance