UNIVERSIDAD AUTÓNOMA DEL ESTADO DE MÉXICO
FACULTAD DE INGENIERÍA

"Desarrollo de un sistema electrónico para el cifrado de señales digitales mediante sistemas dinámicos caóticos, implementado en microcontroladores de bajo costo"

Tesis para obtener el título de

Ingeniero en Electrónica

PRESENTA

Alejandro Bracamonte Hernández

ASESOR

Dr. Javier Salas García

CO-ASESOR

Dra. Laura Luz Valero Conzuelo

Toluca, México. Febrero 2022

1 RESUMEN

La necesidad de proteger la información se ha convertido en uno de los servicios de seguridad más relevantes para la actualidad. El cifrado de información no se encuentra limitado a ejecutarse en sistemas de cómputo. Los actuales sistemas embebidos, como *Internet of Things* (IoT), ejecutan mecanismos de seguridad para cumplir con la confidencialidad del sistema.

Por lo anterior, se planteó el objetivo de desarrollar un sistema electrónico de bajo costo para el cifrado de una señal digital, basado en la teoría de sistemas dinámicos caóticos aplicado a protocolos estandarizados de comunicaciones electrónicas. Cumpliendo la siguiente hipótesis: Es viable construir un sistema electrónico de cifrado digital con la capacidad de funcionar en dispositivos de control de bajo costo.

El prototipo de sistema criptográfico simétrico, objeto del presente trabajo de investigación, obtuvo resultados favorables; el cual se implementó en un microcontrolador ESP32 permitiendo cifrar y descifrar información de forma satisfactoria. De igual manera se realizó un análisis de seguridad mediante la simulación de un ataque; el resultado fue la imposibilidad de lectura de la información confidencial. Por consiguiente, se confirmó la hipótesis del proyecto, asimismo se cumplió con los objetivos planteados.

2 ABSTRACT

The need to protect information has become one of the most relevant security services today. cryptography is not limited to operate on computer systems. Current embedded systems, such as the Internet of Things (IoT), implement security mechanisms to comply with the confidentiality of the system.

Therefore, the objective of developing a low-cost electronic system for the encryption of a digital signal, based on the theory of chaotic dynamic systems applied to standardized protocols of electronic communications, was proposed. Fulfilling the following hypothesis: It is feasible to build an electronic digital encryption system with the ability to function in low-cost control devices.

The prototype of the symmetric cryptographic system obtained good results, which was implemented in an ESP32 microcontroller allowing to encrypt and decrypt information satisfactorily. In the same way, a security analysis was carried out by simulating an attack; the result was the impossibility of reading the confidential information. Therefore, the hypothesis of the project was confirmed, and the objectives set were also met.

5 ÍNDICE GENERAL

1 Resumen 2
2 Abstract 3
3 Agradecimientos 4
4 Dedicatoria 5
5 Índice General 6
Índice de Figuras 9
Índice de Tablas 12
Acrónimos 13
1 Introducción 15
1.1 Panorama general de la seguridad informática en sistemas embebidos. 15
1.2 Introducción a los sistemas de cifrado. 16
1.3 Objetivos. 17
1.4 Hipótesis. 18
1.5 Organización de la tesis 18
2 Antecedentes 20
2.1 Pilares de la seguridad informática. 20
2.2 Modelos de cifrado 21
2.3 Criptografía Ligera 24
2.4 Generación de llaves criptográficas con números pseudoaleatorios. 27

2.5 Atractor de Lorenz como generador de elementos caóticos. 29
2.6 Modelos matemáticos discretos para la resolución de ecuaciones diferenciales ordinarias. 32
3 Materiales y métodos. 34
3.1 Sistema electrónico implementado en el sistema. 34
3.2 Descripción del hardware implementado en el sistema. 35
3.3 Instalación de MicroPython 37
3.4 Programación del ESP32. 38
3.5 Configuración de pines GPIO y comunicación con sensores implementado en MicroPython. 40
3.6 Generación de condiciones iniciales a través del algoritmo SHA-256. 41
3.7 Generación de llave privada mediante valores caóticos generados por el atractor de Lorenz. 44
3.8 Algoritmo criptográfico basado en valores caóticos. 48
4 Resultados y discusión. 49
4.1 Esquemático del diagrama electrónico 49
4.2 Prototipo físico. 52
4.3 Resultados del modelo discreto de Lorenz. 53
4.4 Respuesta del atractor de Lorenz en Salida analógica del convertidor DAC. 55
4.5 Resultados de la lectura y cifrado de datos por parte del emisor. 57
4.6 Resultados de descifrado por parte del receptor. 58
4.7 Simulación de ataque de intercepción. 59
4.8 Resultados del ataque. 60
4.9 Comparación de datos entre atacante y receptor. 66
4.10 Comparación con modelos semejantes. 68
5 Conclusiones 70

5.1 Recomendaciones 71
6 Bibliografía 72
7 Apéndices 74
A. Hardware implementado 74
B. Código main del emisor 75
C. Código main del receptor. 80
D. Código main del atacante. 83
E. Código de la clase LorenzAttractor 86
F. Código de la clase Embedded 93
G. Código de la clase DHT 96
H. Código de la clase accel 97

ÍNDICE DE FIGURAS

Figura 2.1 Principales servicios y mecanismos de seguridad en función a la Norma ITU X.800. Fuente: Elaboración propia. 20
Figura 2.2 Representación del cifrado simétrico entre dos entidades conocidas como emisor y receptor quienes cifran y descifran con una única llave privada compartida previamente. Fuente: Elaboración propia. 22
Figura 2.3 Representación del cifrado asimétrico entre dos entidades conocidas como emisor y receptor. El emisor cifra con la llave pública del receptor. Por su parte el receptor descifra la información con su llave privada. Fuente: Elaboración propia. 23
Figura 2.4 Comparativa de algoritmos criptográficos en función a la velocidad de cifrado y descifrado, memoria requerida y tamaño de la llave. Fuente: Elaboración propia. 24
Figura 2.5 Diagrama de bloques de la generación de llaves privadas mediante datos aleatorios generados por datos semilla. Fuente: Elaboración propia. 27
Figura 2.6 Ejemplos de atractores de Lorenz obtenidos mediante la solución numérica del sistema de ecuaciones diferenciales. Así como la comparación de las variables X, Y, Z de ambos atractores en función al tiempo. Fuente: Elaboración propia. 31
Figura 2.7 Solución numérica y analítica de la ecuación diferencial. Fuente: Elaboración propia. 33
Figura 3.1 Diagrama a bloques del sistema de cifrado implementado en el microcontrolador ESP32. Enfocado en la comunicación entre dos entidades, la lectura de sensores y la visualización en un equipo de cómputo. Fuente: Elaboración propia 34
Figura 3.2 Visualización del puerto COM en el administrador de dispositivos de Windows 10. Fuente: Elaboración propia. 37
Figura 3.3 Diagrama de flujo general del prototipo. Se describen las principales funciones del emisor y del receptor. Se hace énfasis en que ambas entidades comparten características

en común: Lectura de la contraseña, generación de números pseudoaleatorios y generación de llave privada. Fuente: Elaboración propia. 39
Figura 3.4 Diagrama de clases creado para el control del microcontrolador ESP32, los sensores y el sistema de cifrado. Fuente: Elaboración propia. 40
Figura 3.5 Diagrama de flujo del generador de las condiciones iniciales mediante la lectura de una contraseña. Fuente: Elaboración propia. 42
Figura 3.6 Diagrama de flujo de la generación de la llave privada. Consistente en iterar el algoritmo de Runge-Kutta 1024 veces. La llave se genera a través de la operación XOR entre los resultados obtenidos por el algoritmo de Runge-Kutta obteniendo una llave de 1024 bytes. Fuente: Elaboración propia. 47
Figura 3.7 Algoritmo de cifrado simétrico implementado en el microcontrolador ESP32. Fuente: Elaboración propia. 48
Figura 4.1. Diseño esquemático del emisor. Enfocado en la conexión entre el microcontrolador ESP32 y los sensores LM35Z, MPU6050 y DHT22. Asimismo, la implementación del LM324 en configuración de amplificador no inversor para el tratamiento de la señal analógica del sensor LM35Z. Fuente: Elaboración propia. 49
Figura 4.3 Diseño esquemático del receptor, con el protocolo UART habilitado en los pines RX2 y TX2. Fuente: Elaboración propia 50
Figura 4.4 Diagrama esquemático general del emisor y receptor. Fuente: Elaboración propia 51
Figura 4.5 Prototipo desarrollado físicamente en protoboard. Se muestra tanto el emisor en la parte superior, como el receptor en la parte inferior. Fuente: Elaboración propia 52
Figura 4.6 Atractor de Lorenz generado por el ESP32 mediante la digitalización del sistema. Fuente: Elaboración propia 53
Figura 4.7 Resultados del atractor de Lorenz en función al número de iteración generada por el ESP32 y el valor de la variable X, Y, Z de forma descendente. Fuente: Elaboración propia 54
Figura 4.8 Atractor de Lorenz obtenido por las salidas del Convertidor Digital a Analógico (DAC) del ESP32. Comprobando la naturaleza del atractor. Fuente: Elaboración propia. . 55
Figura 4.9 Variables X, Z del Atractor de Lorenz en función al tiempo. Obtenidas por las salidas del convertidor digital a analógico del ESP32. Fuente: Elaboración propia 56

Figura 4.10 Captura de pantalla del software para la visualización de MicroPython en el ESP32. Se muestran los valores de los sensores obtenidos por el microcontrolador. Fuente: Elaboración propia. 57
Figura 4.11 Captura de pantalla del software para la visualización de MicroPython en el ESP32. Se muestran los valores cifrados por el ESP32, mismos que fueron enviados al receptor. Fuente: Elaboración propia. 58
Figura 4.12 Captura de pantalla del software para la visualización de MicroPython en el ESP32. Se muestra la correcta recepción y descifrado de la información por parte del receptor. Fuente: Elaboración propia. 59
Figura 4.13 Diagrama a bloques de la simulación de ataque de intercepción para la verificación de la seguridad del sistema. Fuente: Elaboración propia. 60
Figura 4.14 Diagrama a bloques de los resultados obtenidos en la simulación de ataque al sistema. Es posible observar que los resultados por el atacante son ilegibles. Fuente: Elaboración propia. 61
Figura 4.15 Captura de pantalla del software para la visualización de MicroPython en el ESP32. Lectura por parte del atacante y muestra del correcto funcionamiento del cifrador al no permitir que el atacante pueda leer la información. Fuente: Elaboración propia. 65
Figura 4.16 Resultado del atractor de Lorenz generado por el receptor (Azul) y el atacante (Rojo). Fuente: Elaboración propia. 66
Figura 4.17 Resultados de las variables X, Y, Z generadas por el receptor (Azul) y el atacante (Rojo). Demostrando la confidencialidad del sistema en función a la no repetibilidad por parte del atacante. Fuente: Elaboración propia 67
Figura 4.18 Resultados obtenidos mediante el cifrador desarrollado implementado en el ESP32 (Lorenz ESP) y un equipo de cómputo (Lorenz CPU) comparados con algoritmos criptográficos. Demostrando las ventajas del cifrador caótico. Fuente: Elaboración propia. 68

ÍNDICE DE TABLAS

Tabla 2.1. Comparación entre características de diferentes microcontroladores y un equipo de cómputo promedio. Fuente: Elaboración propia..... 26
Tabla 4.1 Resultados obtenidos en la simulación de ataque, mediante el cambio de un bit en la contraseña establecida por el receptor y el atacante. Demostrando la confidencialidad del sistema. Fuente: Elaboración propia..... 63
Tabla 4.2 Comparación de algoritmos de cifrado con el sistema criptográfico desarrollado mediante el Atractor de Lorenz. Fuente: Elaboración propia..... 69

ACRÓNIMOS

3DES	Triple DES
AES	Advanced Encryption Standard
Blowfish	Algoritmo criptográfico simétricos, diseñado por Bruce Schneier en 1993.
Canal inseguro	En comunicaciones digitales es conocido como cualquier medio de comunicación entre emisor y receptor, puede ser mediante ondas RF, protocolos digitales, internet, etc.
CBC	Cypher Block Chaining
CTR	Counter mode
DDOS	Distributed Denial of Service
DES	Data Encryption Standard
Digesto	También conocido como hash, es el algoritmo matemático que se crea con el fin de poder transformar cualquier conjunto de datos en una serie de caracteres que poseen una longitud fija.
ECB	Electronic Codebook
ECC	*Elliptic curve cryptography*
ESP32	Denominación la familia de chips SoC de bajo costo y consumo de energía desarrollada por Espressif Systems
GPIO	General Pin Input Output
IOT	Internet of things
MAC	Message Authentication Code
MicroPython	Compilador completo del lenguaje Python y un motor e intérprete en tiempo de ejecución, que funciona en el hardware del microcontrolador.

RAM	Random Access Memory
SHA-256	Secure Hash Algorithm of 256 bits
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver Transmitter
XOR	Mejor conocida como compuerta OR exclusiva es una puerta lógica digital cuya salida es verdadera resulta si solo una de las entradas es verdadera.

1 INTRODUCCIÓN

1.1 Panorama general de la seguridad informática en sistemas embebidos.

Durante el año 2021 con el crecimiento inaudito de los sistemas embebidos y su rápida asimilación en la vida diaria de la humanidad, los ataques informáticos a estos sistemas han incrementado a gran velocidad. En 1988 se registró el primer ataque de denegación de servicios, implementado por Robert Tappan Morris (Orman 2003); después de 30 años, los ataques informáticos son más complejos, catastróficos y contrario al sentido común, sencillos de efectuar, por lo cual se presentan con mayor regularidad[2].

Para contrarrestar dicha situación; se constituyó e implementó una serie de normas y protocolos para imposibilitar que los ataques informáticos sean exitosos. Al presente, los servidores de red son los principales objetivos ante ciber-ataques. No obstante, el actual crecimiento en el área de la electrónica aplicada en sistemas embebidos inteligentes como lo es IoT, han generado una nueva área de ataque. Los sistemas embebidos modernos son susceptibles a ataques informáticos de diferentes gamas. En (Lu y Xu 2019) se ilustran los siguientes ataques: Captura de información confidencial, ataques con información maliciosa, malware, virtualización de hilos, phishing, scripts maliciosos, entre otros.

Por lo tanto, es ineludible proteger la información procesada por dichos sistemas. En (Sánchez et al. 1999) se reporta la implementación en un circuito integrado de un criptosistema basado en el esquema de (Al-Hazaimeh et al. 2019). El cual consiste en mejorar

[2] Live cyber-attack map - https://threatmap.checkpoint.com/

la confidencialidad de la información, aplicando protocolos de cifrado analógico. De este modo, solo las entidades autorizadas pueden tener acceso a la información confidencial.

Pese a ello, los actuales modelos de cifrado generan demasiada información. Esta se almacena dentro de una computadora o en su defecto, requieren de múltiples circuitos integrados como memorias. Lo que incita que no sea factible su uso en sistemas embebidos. Por consiguiente, se demanda optimizar este proceso para que sea ejecutado por microcontroladores de bajo costo.

En este sentido, se requiere el desarrollo de un sistema electrónico de cifrado digital que pueda funcionar bajo condiciones de baja capacidad de cómputo, tales como microcontroladores, aunado al hecho de que el sistema sea difícil de "romper" con técnicas de criptoanálisis.

1.2 Introducción a los sistemas de cifrado.

Como se mencionó con anterioridad, el cifrado de la información es el servicio de seguridad informática con mayor relevancia. Para implementarlo se han desarrollado una serie de algoritmos para comunicar información confidencial. Uno de los sistemas más antiguos de los que se tiene registro es el Cifrador de César. El cual se clasifica como un cifrado por sustitución, en el que el alfabeto en el texto plano se desplaza por un número fijo. En (Gowda 2016) se presentan las ventajas y desventajas de este sistema.

El avance de las matemáticas computacionales, así como de la informática permitió realizar sistemas automatizables para mejorar la confidencialidad y seguridad de los algoritmos. En (Abood y Guirguis 2018) se muestran los principales algoritmos criptográficos empleados actualmente. Los cuales hacen uso de llaves privadas y/o públicas indispensables para ejecutar dicho mecanismo de seguridad antes descrito. Estos sistemas criptográficos requieren de una alta capacidad computacional y de almacenamiento para ejecutarse con efectividad. Aunado al hecho de que los ataques informáticos han evolucionado a la par de la seguridad informática, en (Simakov et al. 2018) se describen dos

de los principales problemas de seguridad informática moderna: Meltdown y Spectre, vulnerabilidades de los principales microprocesadores en el mercado durante 2021.

Al presente, los principales sistemas de cifrado, así como los métodos de operación criptográficos (ECB, CBC, CTR) no están diseñados para trabajar en ambientes embebidos. Es indispensable contar con una alta memoria, así como una rápida velocidad de reloj que permita realizar los cómputos en tiempos relativamente cortos. Por ende, se determina que estos sistemas quedan vulnerables ante ataques como: Criptoanálisis por fuerza bruta, robo de llave privada, hombre en el medio, etc.

1.3 Objetivos.

El objetivo principal del presente trabajo consiste en desarrollar un sistema electrónico de bajo costo, para el cifrado de una señal digital, basado en la teoría de sistemas dinámicos caóticos aplicado a protocolos estandarizados de comunicaciones.

El prototipo consiste en una serie de sensores cuya información representa a los datos confidenciales a proteger. Para el desarrollo del sistema de control y procesamiento se emplea el microcontrolador ESP32 codificado en Python enfocado a sistemas embebidos (MicroPython). En este caso se aplica el protocolo de comunicación asíncrona UART (Universal Asynchronous Receiver Transmitter) como medio de comunicación entre entidades. Finalmente, un equipo de cómputo funge como medio de visualización de la información enviada por los microcontroladores.

1.4 Hipótesis.

La hipótesis está basada en las siguientes preguntas de investigación

- ¿Cómo desarrollar un sistema electrónico para el cifrado de una señal digital mediante sistemas dinámicos caóticos e implementarlo en microcontroladores de bajo costo?
- ¿Cómo desarrollar un sistema criptográfico simétrico para microcontroladores de 32 bits?
- ¿Cómo crear llaves privadas en función a los valores del atractor de Lorenz?

Por lo anterior, se estableció la siguiente hipótesis: "***Si se desarrolla un sistema electrónico para el cifrado de una señal digital mediante sistemas dinámicos caóticos, entonces será posible implementarlo en microcontroladores de bajo costo"***

1.5 Organización de la tesis

En **Antecedentes** se muestra el estado del arte referente a los principales temas abordados en el presente trabajo de investigación, de igual forma se desarrolla una breve descripción de los algoritmos criptográficos, métodos numéricos para la solución de ecuaciones diferenciales, así como el comportamiento de sistemas estocásticos como el atractor de Lorenz.

En el capítulo **Materiales y métodos**. se describe la metodología e instrumentación aplicada para la lectura de los sensores analógicos y digitales utilizados en el proyecto objeto de estudio. Al igual que, la descripción del sistema de control, procesamiento de datos y el flujo del software desarrollado, tanto para el control del sistema como para el desarrollo del sistema de cifrado.

En el capítulo **Resultados y discusión**. se describen los datos obtenidos por el atractor de Lorenz de forma digital y analógica. Se mencionan las capacidades del algoritmo de cifrado desarrollado e implementado en microcontroladores de bajo costo. Es de especial interés en este capítulo, la presentación de las derivaciones obtenidas durante la simulación de un ataque al prototipo desarrollado con la intención de verificar la confidencialidad del sistema y su seguridad.

En el capítulo final titulado **Conclusiones**, se presentan de forma resumida los alcances logrados como resultado de diseñar, estructurar, sistematizar y construir el ***algoritmo de cifrado de una señal digital mediante sistemas dinámicos caóticos***, así mismo se incluyen recomendaciones y posibles líneas de investigación a futuro con el propósito de mejorar el comportamiento del sistema.

2 ANTECEDENTES

2.1 Pilares de la seguridad informática.

La seguridad informática es el área relacionada a la protección de la infraestructura computacional, así como la mitigación de las vulnerabilidades relacionadas con las capas de la informática. Las comunicaciones digitales tienden a ser objeto de aplicación para estos servicios.

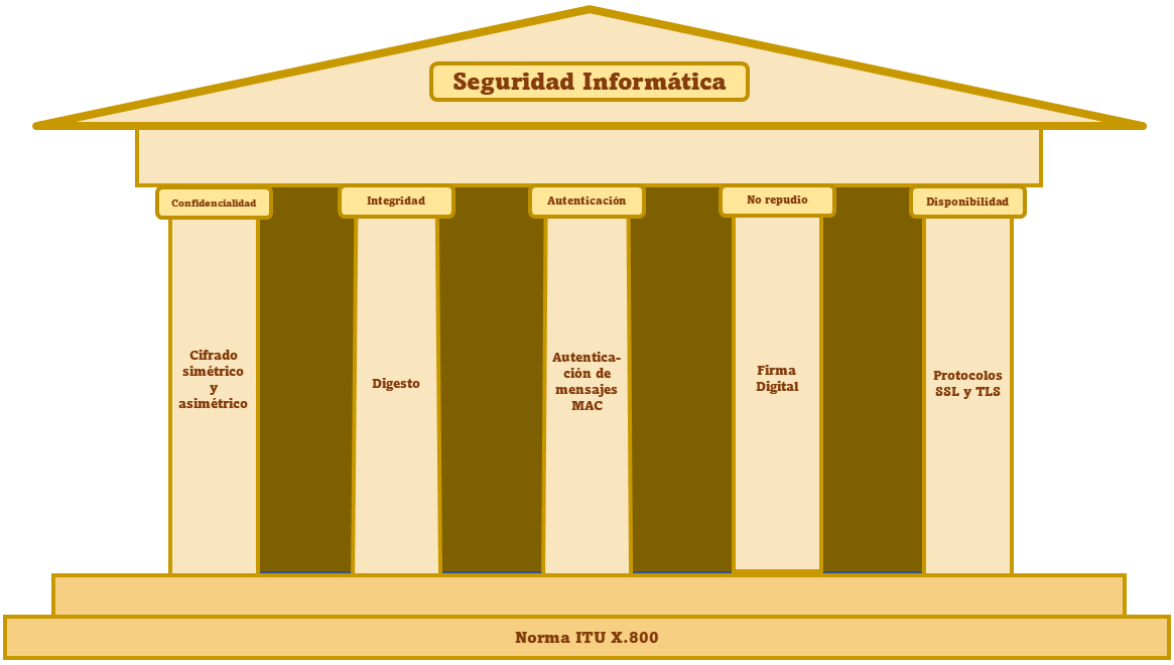


Figura 2.1 *Principales servicios y mecanismos de seguridad en función a la Norma ITU X.800. Fuente: Elaboración propia.*

Como se muestra en la **Figura 2.1** la seguridad informática tiene como base la Norma ITU X.800 *(Union Internacional de Telecomunicaciones 1991)* la cual fue creada como una serie de reglas y recomendaciones a seguir para establecer comunicaciones seguras entre entidades.

Este documento instaura algunos servicios de seguridad que se consideran pilares entre los sistemas de comunicación digital, los cuales son: Confidencialidad, Integridad, Autenticación, No repudio y Disponibilidad. Para desempeñar estos servicios se forman mecanismos de seguridad entre los cuales destacan los siguientes: Cifrado simétrico y asimétrico de información, digesto de algoritmos, autenticación de mensajes MAC, firmas digitales entre entidades y la aplicación de certificados TLS y SSL.

2.2 Modelos de cifrado

El cifrado de información es un mecanismo preciso en la seguridad informática. Para cumplirlo se instituye una serie de algoritmos criptográficos como el cifrador César y la criptografía de curvas elípticas. La criptografía moderna se divide en dos metodologías: Criptografía simétrica y asimétrica.

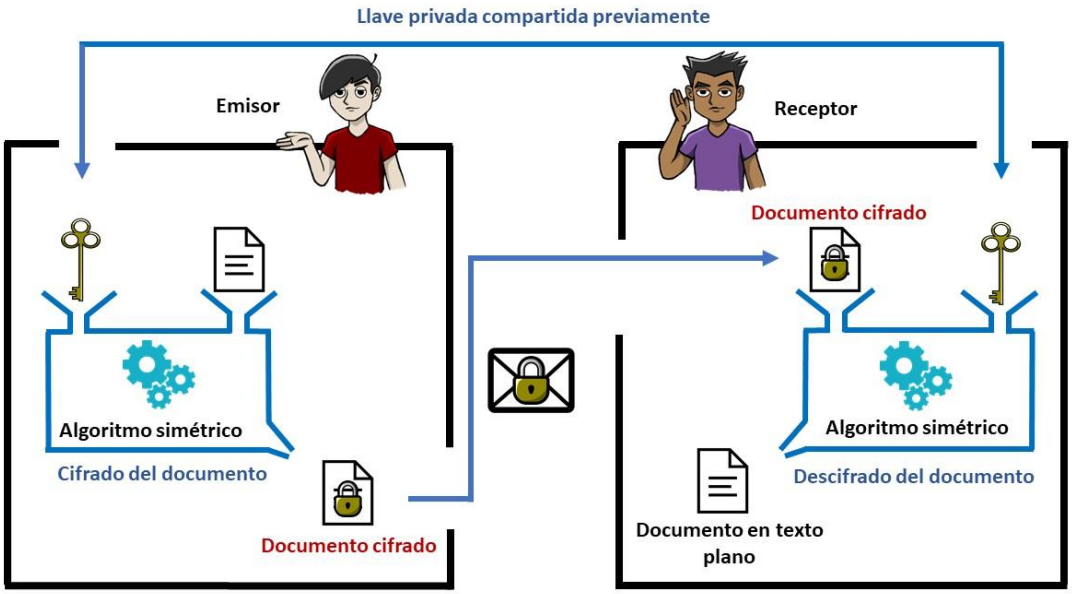


Figura 2.2 *Representación del cifrado simétrico entre dos entidades conocidas como emisor y receptor quienes cifran y descifran con una única llave privada compartida previamente. Fuente: Elaboración propia.*

La criptografía simétrica se basa en una sola llave secreta que ambas entidades comparten. Como se menciona y ejemplifica en (Boneh y Shoup 2015). En la **Figura 2.2** se muestra un proceso de este cifrado. El emisor cuenta con una llave privada con la cual cifra la información (también conocida como datos en blanco) por medio de un algoritmo ya establecido (AES, DES, 3DES, entre otros), generando un documento cifrado. Consecutivamente, envía estos datos al receptor mediante un canal inseguro. El receptor hace uso de la llave privada que comparte con el emisor para descifrar el documento y obtener la información original.

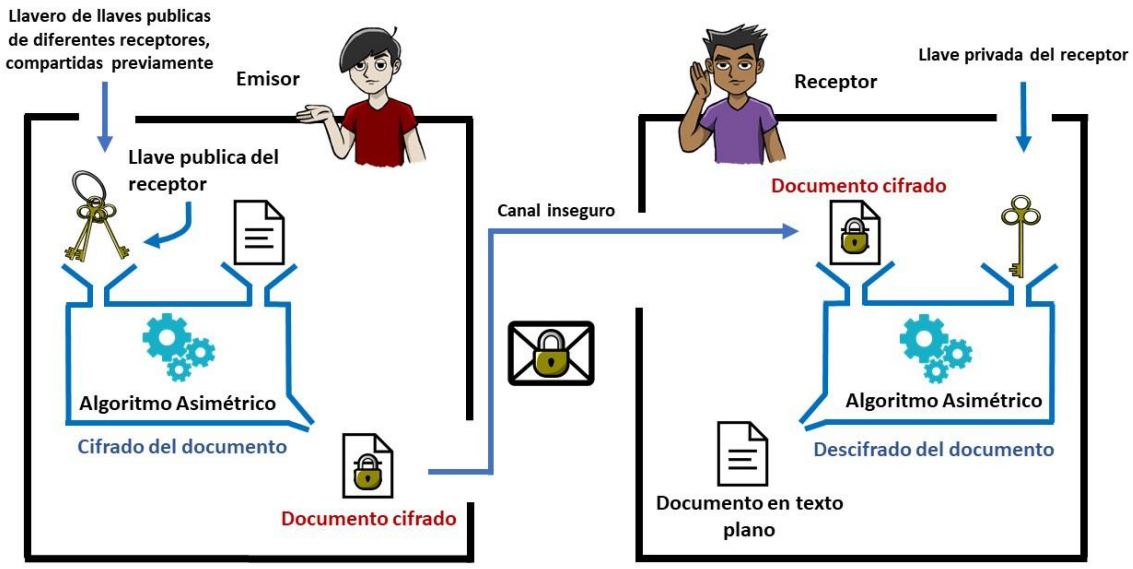


Figura 2.3 *Representación del cifrado asimétrico entre dos entidades conocidas como emisor y receptor. El emisor cifra con la llave pública del receptor. Por su parte el receptor descifra la información con su llave privada. Fuente: Elaboración propia.*

En la criptografía asimétrica cada entidad cuenta con un par de llaves: Pública y privada, como se muestra en la **Figura 2.3**. El emisor tiene en su poder un llavero (conjunto de llaves públicas), en la cual se encuentra la llave pública del receptor. Para cifrar, el emisor hace uso de la llave pública del receptor y de un algoritmo de cifrado (RSA, ECC, entre otros). Se envía la información por un canal inseguro y el receptor descifra la información por medio del mismo algoritmo y con ayuda su llave privada. Ejemplos de lo anterior, así como modelos matemáticos que diferencian a la criptografía de llave pública y privada son mostrados en (Salomaa 2013).

2.3 Criptografía Ligera

Como se expuso en el capítulo Introducción, la necesidad de sistemas criptográficos es indiscutible. Razón por la cual existe una serie de algoritmos criptográficos implementados en equipos con un poder de cómputo alto.

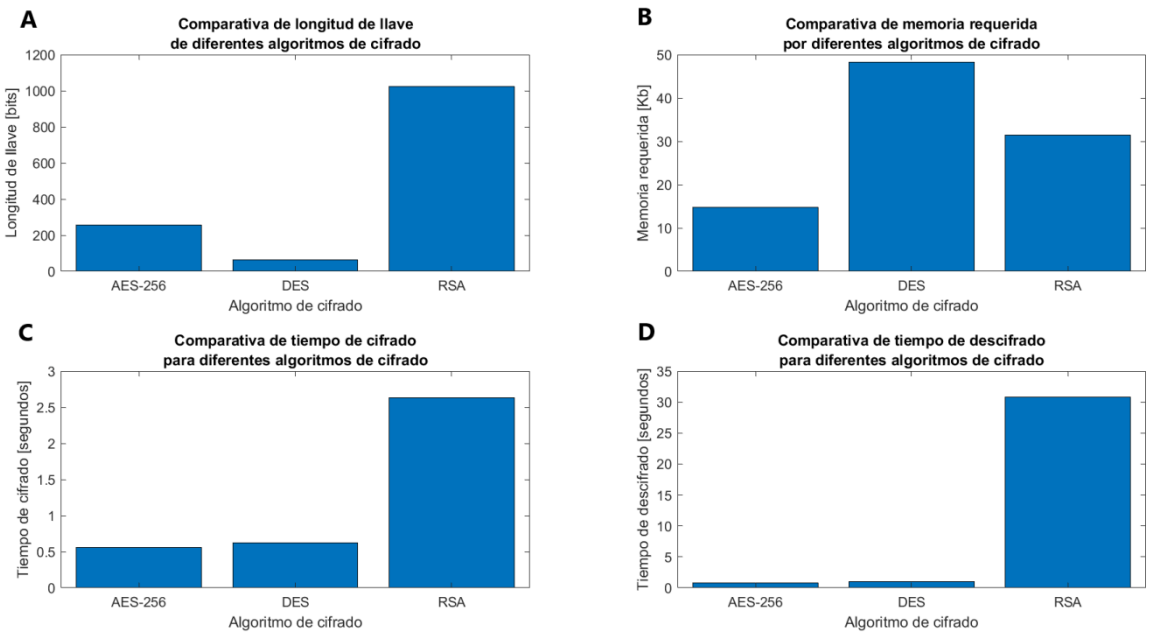


Figura 2.4 *Comparativa de algoritmos criptográficos en función a la velocidad de cifrado y descifrado, memoria requerida y tamaño de la llave. Fuente: Elaboración propia.*

En la **Figura 2.4** se muestra una comparación de las principales características entre diferentes algoritmos de cifrado (DES, AES-256 y RSA). En la **gráfica A** se representa la longitud de la llave requerida por los algoritmos en función a su tamaño (bits), siendo 1024 bits el mayor tamaño requerido por el algoritmo RSA. En cuanto a la **gráfica B** se representa el tamaño (kilobytes) requerido por parte del algoritmo, es decir, el espacio de memoria que el algoritmo en sí mismo ocupa dentro del sistema. En cuanto a memoria el algoritmo DES requiere una cantidad mayor para su instalación. Finalmente, las **gráficas C y D** representan el tiempo de cifrado y descifrado respectivamente, en otras palabras, el tiempo promedio que el algoritmo requiere para cifrar la información y descifrarla, siendo el algoritmo RSA el más lento en ambos casos.

En (Hercigonja 2016) se puede observar una comparativa a mayor profundidad de estos algoritmos. Es rescatable mencionar que dicha información fue desarrollada haciendo uso de equipos de cómputo promedio. Los cuales cuentan con una diferencia abismal en función a sus características con los microcontroladores. En la **Tabla 2.1** se puede observar esta diferencia, demostrando que el uso de sistemas criptográficos típicos no es factible para un sistema embebido.

De aquí la importancia de la criptografía ligera, la cual se basa en la contraposición de los requisitos computacionales necesarios para ejecutar los algoritmos de cifrado más comunes (AES, DES, RSA, ECC, Blowfish, etc), eficientizando los códigos para poder trabajar bajo elementos de bajos recursos tales como los microcontroladores antes mencionados. Haciendo un especial énfasis en el manejo de memoria RAM para el almacenamiento de variables y ROM para el almacenamiento del algoritmo, además de funcionar con una frecuencia de reloj del orden de los MHz o menor.

Tabla 2.1. *Comparación entre características de diferentes microcontroladores y un equipo de cómputo promedio[3]. Fuente: Elaboración propia.*

	Velocidad de reloj [MHz]	**Capacidad de almacenamiento [MB]**	**Memoria RAM [MB]**	**Potencia [W]**
Equipo de Cómputo promedio	1 000	250 000	2 000	60
Raspberry Pi 4	1 500	16 000	4 000	15
Raspberry Pico	133	16	2	1.24
ESP32	240	4	0.520	0.5
MSP430G2553	16	.016	.512	0.01

[3] Las características promedio del equipo se muestran en la sección **3.2 Descripción del hardware implementado en el sistema.**

2.4 Generación de llaves criptográficas con números pseudoaleatorios.

En (Aumasson 2017) se menciona la importancia de los números aleatorios para la criptografía. Sin embargo, la generación de elementos con aleatoriedad sigue siendo un problema para el actual procesamiento digital de la información, por lo tanto, se han desarrollado una serie de algoritmos con la capacidad de generar números pseudo aleatorios, mediante una información semilla.

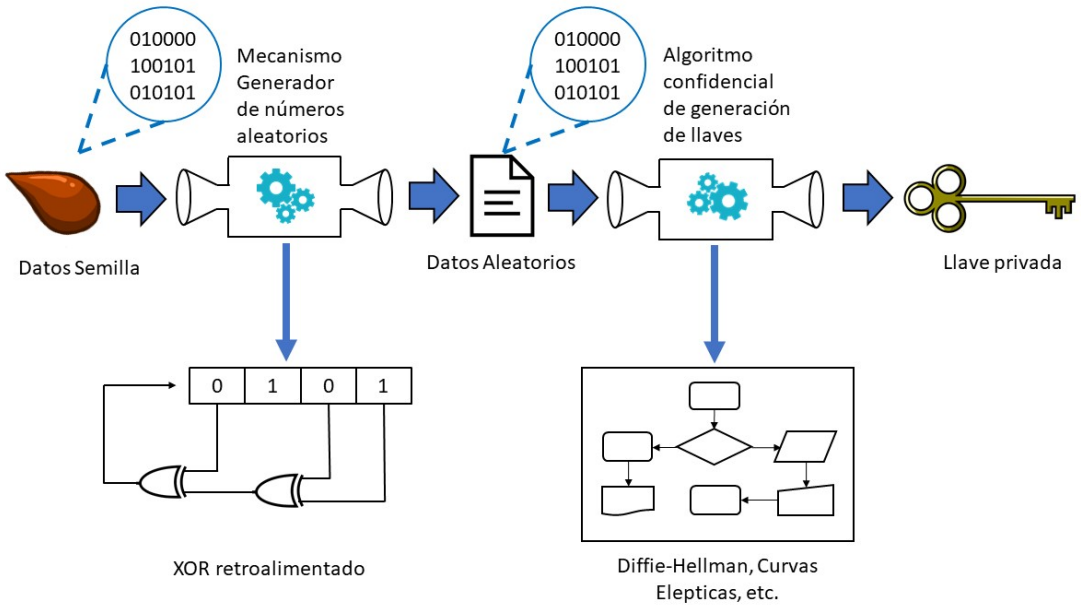


Figura 2.5 *Diagrama de bloques de la generación de llaves privadas mediante datos aleatorios generados por datos semilla. Fuente: Elaboración propia.*

Los métodos de cifrado requieren de una llave. En función al método de cifrado el algoritmo de generación de llave es diverso. En la

Figura 2.5 Figura 2.5 se muestra de forma general el proceso para la generación de llaves privadas.

Se requiere de una serie de bits aleatorios que funcionen como "semilla" del algoritmo. En (Aumasson 2017) se muestra un método común, el Registro de corrimiento lineal con retroalimentación LFSR (por sus siglas en inglés *"Linear Feedback Shift Register"*) el cual cuenta con una cadena inicial de bits, con una retroalimentación por medio de compuertas XOR, como se muestra en la **Figura 2.5**. En cada ciclo de reloj, se obtiene un bit aleatorio en función al registro. Teniendo como tamaño máximo un valor de 2^n bits, donde n es el tamaño del registro inicial.

Una vez que se obtienen dichos números pseudo aleatorios, estos ingresan a un algoritmo de generación de llaves, el cual hace un tratamiento especial a los bits para la generación de esta. Por ejemplo, al aplicar un logaritmo discreto a los bits da como resultado la llave privada. En (Aumasson 2017) se presentan una serie de algoritmos para la generación de llaves por medio del algoritmo de Diffie-Hellman.

2.5 Atractor de Lorenz como generador de elementos caóticos.

En 1979, el Dr. Edward Lorenz modelaba el comportamiento del clima a través de una serie de ecuaciones diferenciales, con el objetivo de predecir el comportamiento de variables como: Velocidad del viento, temperatura ambiental y presión barométrica. Como resultado obtuvo un modelo matemático que se muestra en el sistema de ecuaciones diferenciales de la **Ecuación (2.1)**

$$\begin{aligned} \frac{dx}{dt} &= -\sigma x + \sigma y \\ \frac{dy}{dt} &= -xz + \gamma x - y \\ \frac{dz}{dt} &= xy - \beta z \end{aligned} \tag{2.1}$$

En (Ghys 2013) se expresa una interpretación física del modelo de Lorenz. Donde x representa la intensidad de la convección, y representa la diferencia de temperatura entre las corrientes ascendentes y descendentes, y z es proporcional a la distorsión del perfil de temperatura vertical de la linealidad, un valor positivo indica que los gradientes más fuertes ocurren cerca de los límites. La constante σ es el número de Prandtl. Generalmente para establecer un atractor y converger, se aplican las siguientes constantes: $\gamma = 28$, $\sigma = 10$ y $\beta = \frac{8}{3}$.

Tras realizar los cálculos matemáticos y graficar los resultados se dispuso a realizar el mismo procedimiento mediante cálculos computacionales. El resultado fue una diferencia radical en los datos. A pesar de que se modelo el mismo sistema con las mismas condiciones iniciales, los resultados no eran iguales. El error radicaba en la forma en que la computadora comprende los números.

Por ejemplo, el número racional $\frac{1}{3}$ es considerado una serie infinita de números decimales. Pero al momento de ingresar y guardar la fracción en un registro según el standard IEEE 854[4], se almacena como 1.3333333333333333 (0x3FF5555555555555 en formato hexadecimal) es fácil notar el truncado de los decimales. Obteniendo un error menor a la millonésima parte. Sin embargo, esta diferencia es suficiente para alterar el comportamiento de los resultados.

Con ello se descubrió el comportamiento de los sistemas determinísticos altamente sensibles a las condiciones iniciales. En (Morón 2020) se establece que un sistema dinámico es aquel cuyo estado evoluciona con el tiempo. Un sistema dinámico es determinista si las reglas de evolución que lo rigen determinan de manera unívoca su estado futuro a partir del conocimiento del estado presente. En cuanto este sistema presenta un comportamiento aperiódico e irregular, se conoce como caos determinista, con lo anterior se establecieron las bases de la teoría del caos.

[4] https://standards.ieee.org/standard/854-1987.html

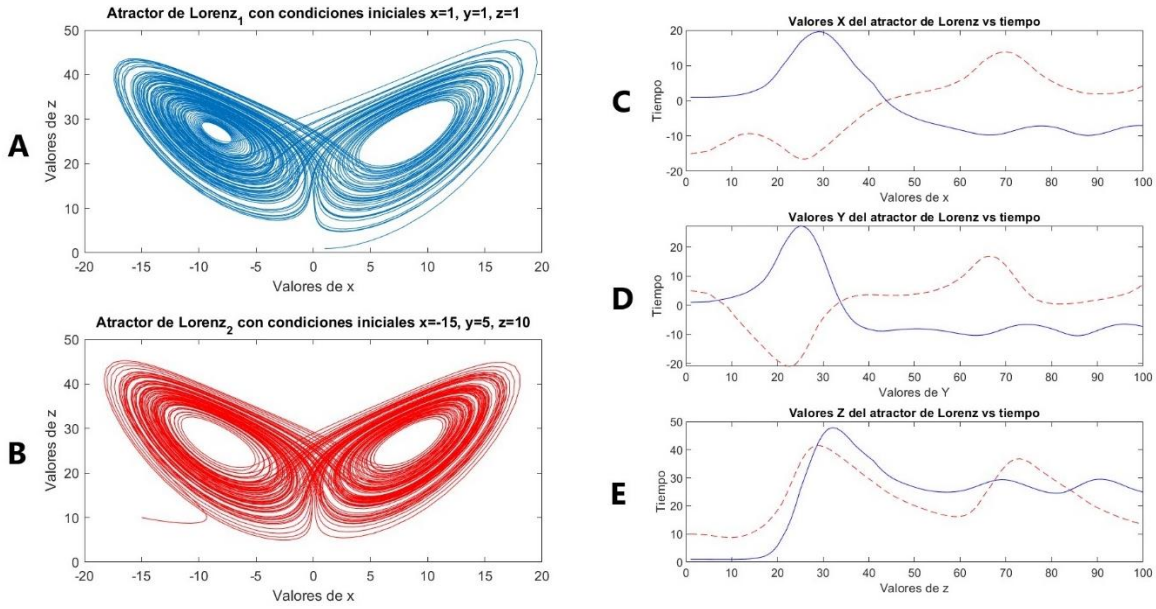


Figura 2.6 *Ejemplos de atractores de Lorenz obtenidos mediante la solución numérica del sistema de ecuaciones diferenciales.* Así como la comparación de las variables X, Y, Z de ambos atractores en función al tiempo. *Fuente: Elaboración propia.*

En la **Figura 2.6** se puede visualizar de forma gráfica el resultado de realizar los cálculos para el atractor de Lorenz con 2 diferentes grupos de condiciones iniciales. Para el atractor A (azul) se consideró el uso de $(X_0, Y_0, Z_0) = (1,1,1)$, mientras que para el atractor B (rojo) se consideró $(X_0, Y_0, Z_0) = (-15,5,10)$. Se puede observar en las **Gráficas A** y **B** de la **Figura 2.6** que los resultados, aunque caóticos, tienden a formar una figura estable conocida como atractor extraño, lo que nos permite tener cierto control sobre el sistema. Aunado a lo anterior, en las **Gráficas C, D** y **E** se puede observar la diferencia entre los valores obtenidos en función al tiempo para cada variable. Demostrando la no repetibilidad de los datos y la alta sensibilidad a las condiciones iniciales.

2.6 Modelos matemáticos discretos para la resolución de ecuaciones diferenciales ordinarias.

La compresión natural del ser humano frente a los modelos matemáticos continuos difiere del modelo discreto que maneja cualquier sistema digital. Por ende, para obtener soluciones numéricas de funciones matemáticas es necesario realizar iteraciones computaciones por medio de algoritmos.

En (Press et al. 2007) se muestra el método de Runge-Kutta que proporciona una solución matemática al evaluar a la función y hacer uso de una pendiente para predecir el siguiente punto dentro de la gráfica. El conjunto de ecuaciones establecidas para este algoritmo se muestra en la **Ecuación (2.2)**

$$\frac{dy}{dx} = f(x,y), y(x_0) = x_0$$

$$y_{k+1} = y_k + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

$$\begin{cases} k_1 = f(x_k, y_k) \\ k_2 = f\left(x_k + \frac{h}{2}, y_k + \frac{k_1 h}{2}\right) \\ k_3 = f\left(x_k + \frac{h}{2}, y_k + \frac{k_2 h}{2}\right) \\ k_4 = f(x_k + h, k_3 h) \end{cases} \tag{2.2}$$

El algoritmo es expresado como una iteración de valores a través de la función $y_{k+1} = y_k + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$ también conocida como pendiente calculada, aplicada a un punto inicial $y(x_0) = x_0$ donde los factores k_1, k_2, k_3, k_4 están definidos por las evaluaciones

de la función $\frac{dy}{dx} = f\left(x, y\right)$ mediante el auto incremento de la variable independiente en un tamaño h también conocido como “paso”.

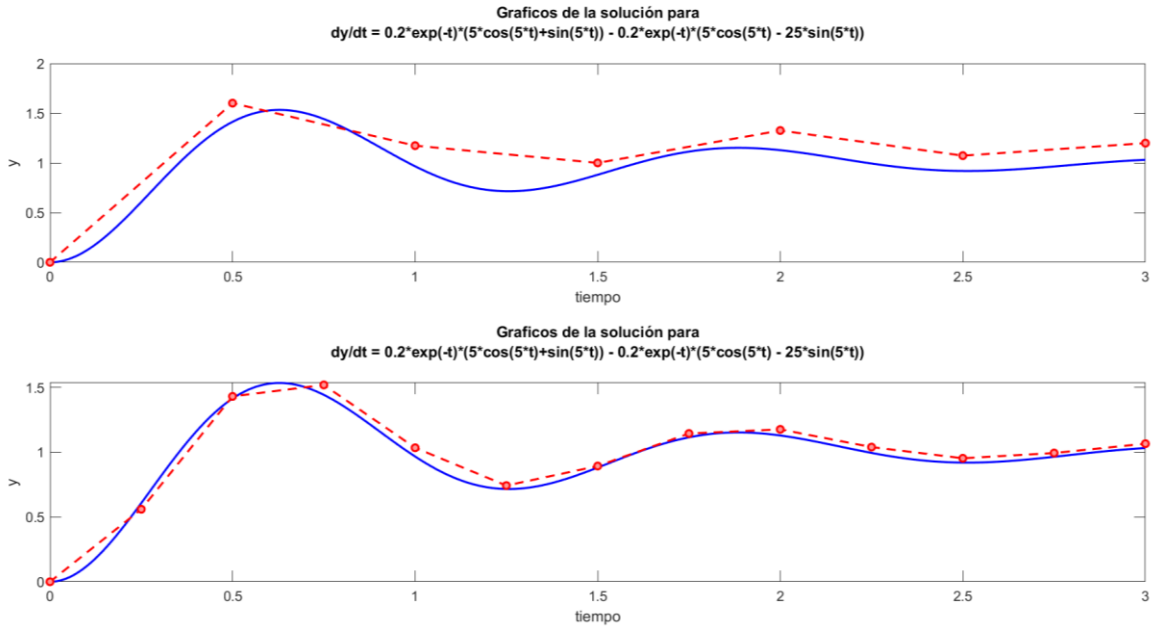


Figura 2.7 *Solución numérica y analítica de la ecuación diferencial. Fuente: Elaboración propia.*

En la **Figura 2.7** se puede observar el comportamiento de la ecuación diferencial de primer grado $\frac{dy}{dt} = 0.2e^{-t}\left(5\cos(5t) + \sin(5t) - 5\cos(5t) + 25\sin(5t)\right)$ tanto de forma analítica como numérica por medio del método de Runge-Kutta. En este caso en particular se puede apreciar la diferencia entre los resultados analíticos y la aproximación numérica. Para el primer caso se estableció a h = 0.5 mientras que el segundo caso h = 0.25. Es fácil deducir que entre más pequeño sea h, la aproximación es más precisa.

3 MATERIALES Y MÉTODOS.

3.1 Sistema electrónico implementado en el sistema.

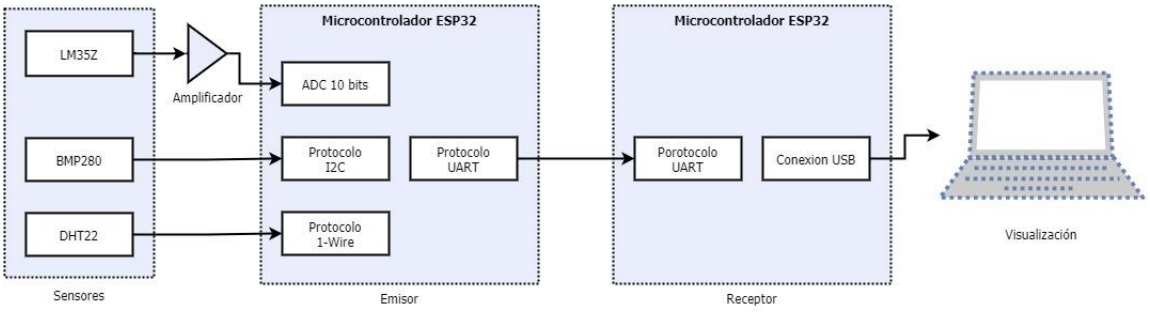


Figura 3.1 *Diagrama a bloques del sistema de cifrado implementado en el microcontrolador ESP32. Enfocado en la comunicación entre dos entidades, la lectura de sensores y la visualización en un equipo de cómputo. Fuente: Elaboración propia*

Como se muestra en la **Figura 3.1** el prototipo del sistema se comporta bajo el siguiente esquema: Se cuenta con un sistema de sensores digitales y analógicos para la lectura de la temperatura ambiental y la posición relativa del prototipo. Mismos que se encuentran conectados al microcontrolador ESP32 por medio de protocolos digitales de comunicación estandarizada y la lectura del Convertidor Analógico a Digital interno del ESP32.

Por parte del receptor se cuenta con un segundo microcontrolador ESP32 que se conectó al emisor por medio del protocolo UART a 9600 baudios[5]. Para la visualización de datos se realizó la conexión por medio del puerto serial a un equipo de cómputo. Aunado a lo anterior fue necesario implementar MicroPython y la instalación del entorno de desarrollo PyCharm[6] en el equipo de cómputo para visualizar los datos enviados mediante USB.

3.2 Descripción del hardware implementado en el sistema.

A continuación, se describen las características generales de las unidades de control, sensores y visualización previamente descritos en el diagrama a bloques.

Unidad de Control

- Microcontrolador ESP32.
- Firmware: MicroPython
- CPU: Microprocesador de 32-bit de doble núcleo, 240 MHz.
- Memoria: 520 KiB SRAM
- 12-bit SAR ADC
- Protocolos de comunicación.
    - 4 canales SPI
    - 2 canales I²S
    - 2 canales I²C
    - 3 canales UART

Sensores

- LM35Z
    - Sensor de temperatura ambiental analógico.

[5] Unidad de medida de la velocidad de transmisión de señales que se expresa en símbolos por segundo.
[6] PyCharm: the Python IDE for Professional Developers by JetBrains - https://www.jetbrains.com/pycharm/

- Voltaje: 3V - 20V
- Resolución: 1 °C/10mV
- Rango: -55°C - 150°C.

- DH22
    - Sensor de temperatura y humedad digital.
    - Voltaje: 3.3V – 6.0V
    - Interfaz: 1-Wire
    - Rango de temperatura: -40°C - 80°C.
    - Rango de Humedad: 0% - 100%

- MPU6050
    - Acelerómetro de 3 ejes.
    - Voltaje: 3.3V - 5V
    - Grados de libertad: 6
    - Rango Acelerómetro: 2g/4g/8g/16g
    - Rango Giróscopio: 250Grad/s, 500Grad/s, 1000Grad/s, 2000Grad/s
    - Interfaz: I2C
    - Conversor ADC: 16 Bits (salida digital)

Equipo de Cómputo

- Procesador: AMD A8-6410 APU con AMD Radeon R5 Graphics 2.00 GHz
- Memoria RAM: 6.00 GB (4.93 GB utilizable)
- Tipo de sistema: Sistema operativo Windows 10 de 64 bits, procesador x64
- Software: PyCharm
- Firmware: MicroPython

3.3 Instalación de MicroPython

Previo a la programación del microcontrolador ESP32, fue necesario la configuración e instalación del firmware de MicroPython en el microcontrolador ESP32. En la página oficial de MicroPython[7] se puede encontrar una metodología de instalación. Para lo cual fue necesario ejecutar los siguientes comandos dentro del símbolo del sistema. Recordando que fue indispensable tener instalado Python 3.x en el equipo de cómputo que fungió como sistema general.

```
$ python -m pip install esptool
```

Para proseguir con la configuración e instalación, se verificó el puerto COM al que se encontraba vinculada el microcontrolador ESP32 como se muestra en la **Figura 3.2**.

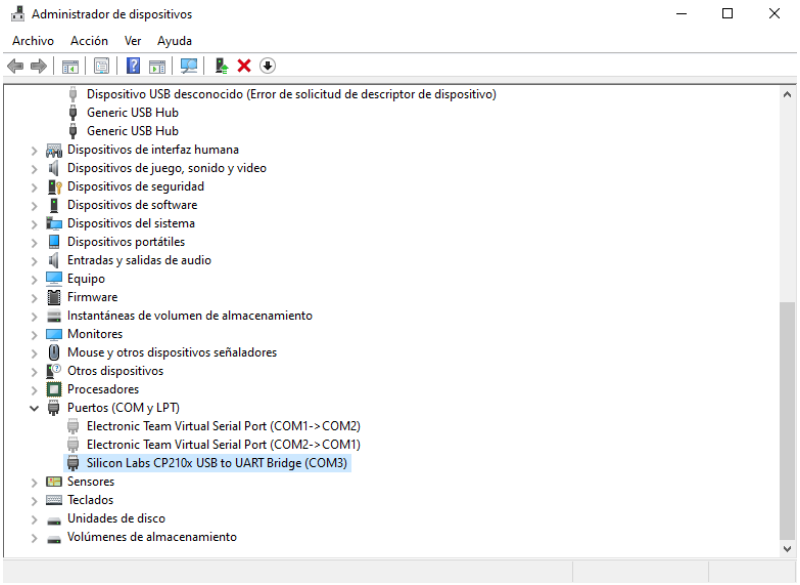


Figura 3.2 *Visualización del puerto COM en el administrador de dispositivos de Windows 10. Fuente: Elaboración propia.*

[7] https://micropython.org/

El firmware de MicroPython se puede encontrar en la página oficial de MicroPython, en la cual se encuentra una serie de archivos binarios. Se optó por la utilización de esp32-idf4-20200902-v1.13.bin por ser la última versión estable diseñada para el ESP32, para la fecha en la que se escribió el presente texto.

Con la información obtenida de la **Figura 3.2** se concluyó con los siguientes comandos para formatear la memoria ROM del ESP32 y realizar la instalación del firmware (esp32-idf4-20200902-v1.13.bin).

```
$ esptool.py --chip esp32 --port COM3 erase_flash

$ esptool.py --chip esp32 --port COM3 --baud 460800 write_flash -z 0x1000
esp32-idf4-20200902-v1.13.bin
```

3.4 Programación del ESP32.

El microcontrolador ESP32 está diseñado para trabajar en dos partes: La sección de configuración, ejecutada una sola vez, y el ciclo infinito, donde se encuentran las sentencias que serán ejecutadas continuamente mientras el sistema se encuentre encendido.

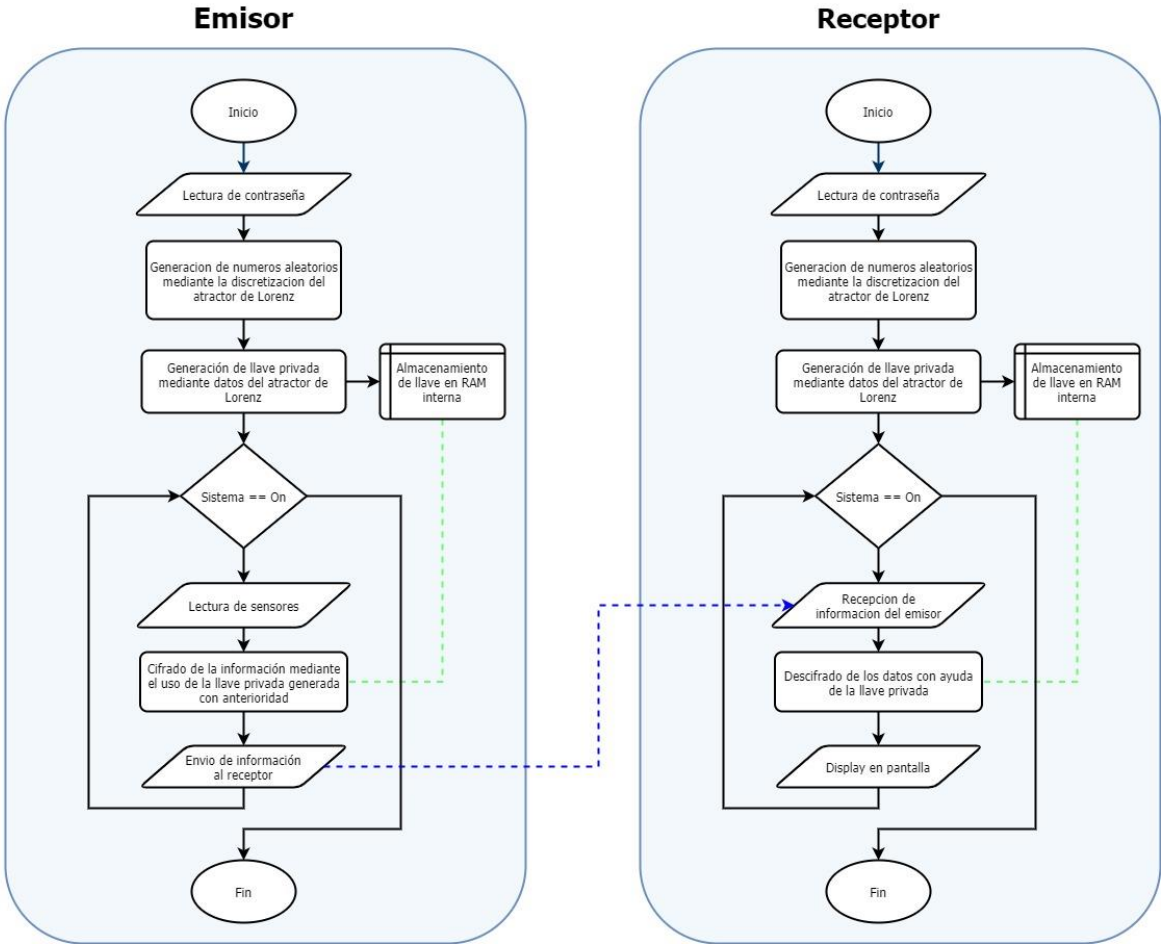


Figura 3.3 *Diagrama de flujo general del prototipo. Se describen las principales funciones del emisor y del receptor. Se hace énfasis en que ambas entidades comparten características en común: Lectura de la contraseña, generación de números pseudoaleatorios y generación de llave privada. Fuente: Elaboración propia.*

En la **Figura 3.3** se muestra el diagrama de flujo de ambas entidades. Se basó en la lectura de la contraseña y la generación del digesto de esta. Una vez obtenidos, estos bytes fungieron como los elementos pseudo - aleatorios necesarios para la generación de la llave, misma que se almacenó en la memoria RAM de las entidades.

En cuanto al bucle infinito el emisor realizó una lectura de los sensores, posteriormente se cifró la información y se envió por el protocolo UART, repitiéndose este ciclo. En cuanto el receptor, este espero la información obtenida y la descifró, una vez concluido el proceso anterior se mostraron los resultados en pantalla, en este caso en el equipo de cómputo.

3.5 Configuración de pines GPIO y comunicación con sensores implementado en MicroPython.

Dado que el firmware usado es enfocado en MicroPython, un lenguaje orientado a objetos, se desarrolló el sistema en función a este paradigma. Se dividió el sistema completo en pequeños subsistemas modelados en una clase y las acciones de este, como la lectura del *Analogic to Digital Converter* (ADC) forma parte de los métodos programados para la clase.

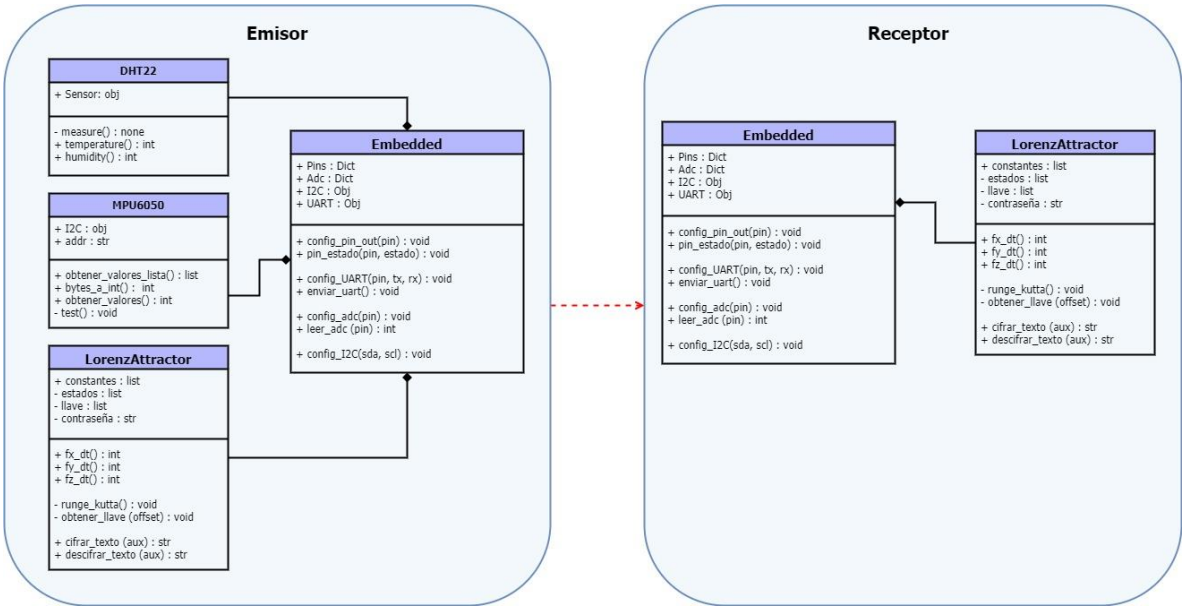


Figura 3.4 *Diagrama de clases creado para el control del microcontrolador ESP32, los sensores y el sistema de cifrado. Fuente: Elaboración propia.*

En la **Figura 3.4** se observa el diagrama de clases a partir del cual se programaron las diferentes instancias del sistema. La clase principal que corresponde a *Embedded* fue la encargada de controlar todas las características físicas del ESP32, tales como: Configuraciones de *General Purpose Input/Output* (GPIO), inicialización de protocolos de comunicación, lectura de sensores, etc.

Por su parte la clase *LorenzAttractor* formó parte del sistema de cifrado desarrollado durante el proyecto, esta clase contiene los métodos para la generación de la llave, cifrado y descifrado de mensajes. Las clases *MPU6050* y *DHT22* fueron exclusivas del emisor al ser el encargado de la lectura de los sensores. Aunado a lo anterior, en la **Figura 3.4** se muestra la conexión realizada entre las clases, donde se hizo uso de los métodos de composición para interacción entre clases.

3.6 Generación de condiciones iniciales a través del algoritmo SHA-256.

Como se mencionó en **Antecedentes**, la generación de números aleatorios fue un paso indispensable para el sistema criptográfico. Se tomó especial interés en que la entrada de información por parte del usuario fuese sencilla de recordar por el mismo. Por consiguiente, se optó por tomar como base una contraseña en formato *string* ingresada directamente en el código, como elemento inicial para la generación de números aleatorios.

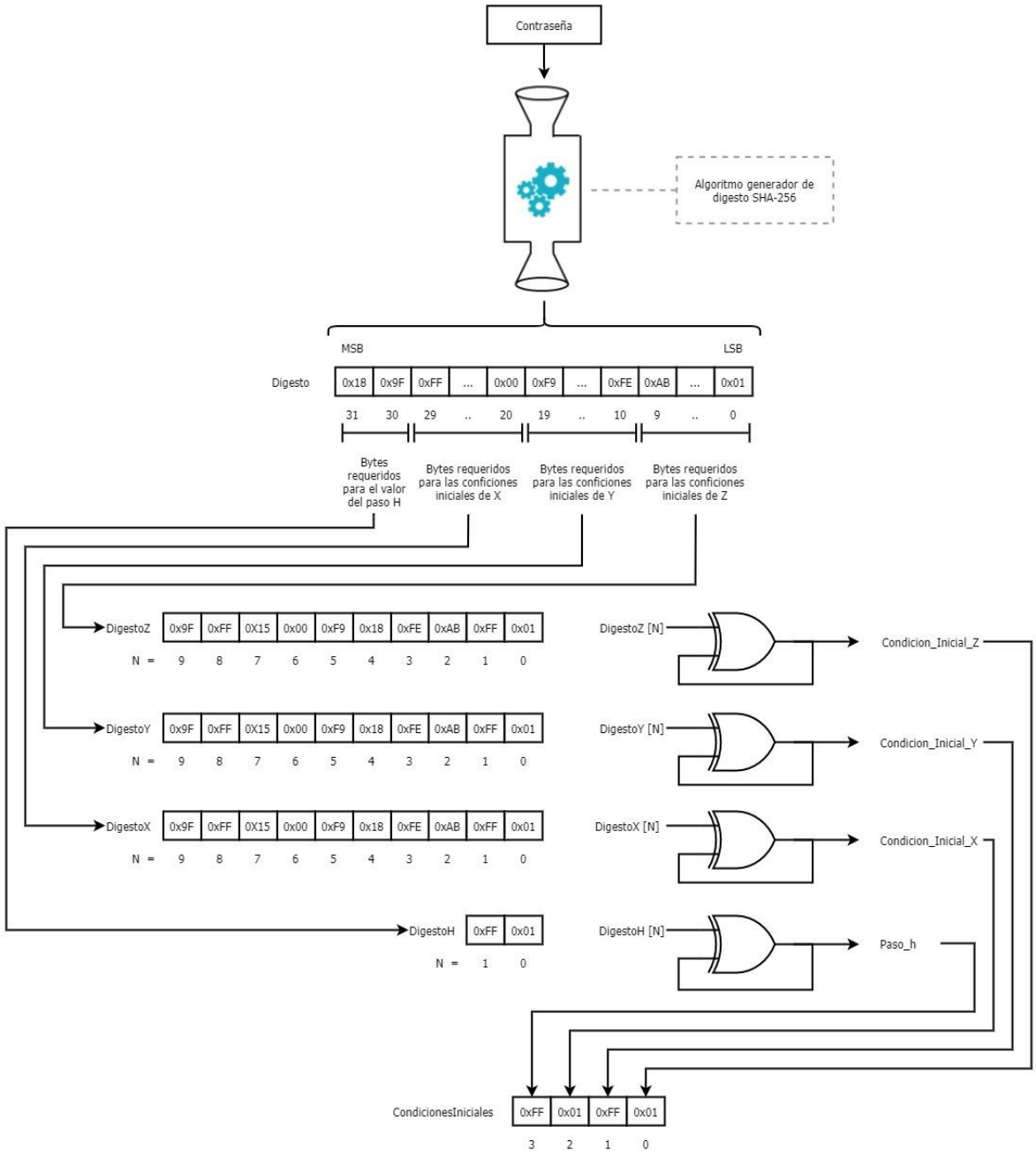


Figura 3.5 *Diagrama de flujo del generador de las condiciones iniciales mediante la lectura de una contraseña. Fuente: Elaboración propia.*

En la **Figura 3.5** se muestra el proceso para la inicialización de las condiciones iniciales necesarias que fungieron como elemento semilla para la generación de números aleatorios. El primer paso fue la obtención del digesto por medio del algoritmo SHA-256, obteniendo una cadena de 32 bytes, la cual no es sensible al tamaño del dato de entrada.

Para aumentar la seguridad del algoritmo criptográfico y mejorar la sensibilidad al cambio de bits, se desarrolló un sistema de adecuamiento de registros el cual consiste en separar la cadena de bytes en 4 paquetes: 3 cadenas de 10 bytes correspondientes a las condiciones iniciales de *X, Y, Z* y una cadena de 2 bytes asignada al paso *h* requerido en el algoritmo de Runge-Kutta.

Finalmente se realizó una serie de operaciones *XOR* retroalimentados entre cada uno de los bytes de las cadenas *DigestoX*, *DigestoY*, *DigestoZ* y *DigestoH*. El resultado se almacenó en una nueva cadena denominada: *CondicionesIniciales*, como se observa en la parte inferior de la **Figura 3.5**.

3.7 Generación de llave privada mediante valores caóticos generados por el atractor de Lorenz.

Una vez obtenidas las condiciones iniciales, se procedió a realizar la digitalización del atractor de Lorenz por medio del algoritmo de Runge-Kutta de cuarto grado, para lo cual se modificó el algoritmo descrito en el capítulo **Antecedentes** para adecuarlo a la solución de un sistema de ecuaciones diferenciales de 3 variables. La **Ecuación (3.1)**. muestra el resultado del siguiente valor para las 3 variables inmersas en el atractor.

$$\begin{aligned} x_{k+1} &= x_k + \frac{\left(k_{x1} + 2k_{x2} + 2k_{x3} + k_{x4}\right)}{6} \\ y_{k+1} &= y_k + \frac{\left(k_{y1} + 2k_{y2} + 2k_{y3} + k_{y4}\right)}{6} \\ z_{k+1} &= z_k + \frac{\left(k_{z1} + 2k_{z2} + 2k_{z3} + k_{z4}\right)}{6} \end{aligned} \tag{3.1}$$

Dado el sistema de ecuaciones antes mencionado, fue necesario definir los valores de *k* respecto a las funciones matemáticas de la **Ecuación (3.2)** expresadas en el capítulo **Antecedentes**.

$$\begin{aligned} k_1 &= \frac{df}{dt}\left(x_k, y_k, z_k\right) \\ k_2 &= \frac{df}{dt}\left(x_k + \frac{k_{x1}h}{2}, y_k + \frac{k_{y1}h}{2}, z_k + \frac{k_{z1}h}{2}\right) \\ k_3 &= \frac{df}{dt}\left(x_k + \frac{k_{x2}h}{2}, y_k + \frac{k_{y2}h}{2}, z_k + \frac{k_{z2}h}{2}\right) \\ k_4 &= \frac{df}{dt}\left(x_k + k3, y_k + k_{y3}, z_k + k_{z3}\right) \end{aligned} \tag{3.2}$$

Posteriormente se desarrolló el código del método para la solución numérica del atractor de Lorenz. Codificado para ser funcional en Python y MicroPython por igual. Para continuar con los elementos de seguridad en sistemas embebidos, se prestó especial interés en realizar el encapsulamiento de las clases, aplicando la protección a métodos para impedir que el usuario tenga acceso a funciones no indispensables. De igual forma, se estableció el método Runge-Kutta como un método estático para mejorar la velocidad del sistema.

```
    def __fx(self, x, y, z) -> float:
        dxdt = self.constants[1] * (y - x)  # Sigma
        return dxdt


    def __fy(self, x, y, z) -> float:
        dydt = x * (self.constants[0] - z) - y  # RHo
        return dydt


    def __fz(self, x, y, z) -> float:
        dzdt = x * y - self.constants[2] * z  # Beta
        return dzdt


    @staticmethod
    def __runge_kutta(self, x, y, z, h) -> list:
        k1x = h * self.__fx(x, y, z)
        k1y = h * self.__fy(x, y, z)
        k1z = h * self.__fz(x, y, z)
```

```
        k2x = h * self.__fx(x + k1x / 2, y + k1y / 2, z + k1z / 2)
        k2y = h * self.__fy(x + k1x / 2, y + k1y / 2, z + k1z / 2)
        k2z = h * self.__fz(x + k1x / 2, y + k1y / 2, z + k1z / 2)

        k3x = h * self.__fx(x + k2x / 2, y + k2y / 2, z + k2z / 2)
        k3y = h * self.__fy(x + k2x / 2, y + k2y / 2, z + k2z / 2)
        k3z = h * self.__fz(x + k2x / 2, y + k2y / 2, z + k2z / 2)

        k4x = h * self.__fx(x + k3x, y + k3y, z + k3z)
        k4y = h * self.__fy(x + k3x, y + k3y, z + k3z)
        k4z = h * self.__fz(x + k3x, y + k3y, z + k3z)

        kx = x + ((k1x + 2 * k2x + 2 * k3x + k4x) / 6)
        ky = y + ((k1y + 2 * k2y + 2 * k3y + k4y) / 6)
        kz = z + ((k1z + 2 * k2z + 2 * k3z + k4z) / 6)

        return [kx, ky, kz, int(kx) ^ int(ky) ^ int(kz)]
```

Finalmente se implementó el algoritmo dentro de un ciclo para generar los valores que formaron parte del atractor de Lorenz discretizado. Como se muestra en la **Figura 3.6** la llave se generó tras iterar 1024 veces el algoritmo de Runge-Kutta y realizar una serie de operaciones XOR entre los bytes obtenidos, el resultado se almacenó dentro de los registros de la llave privada.

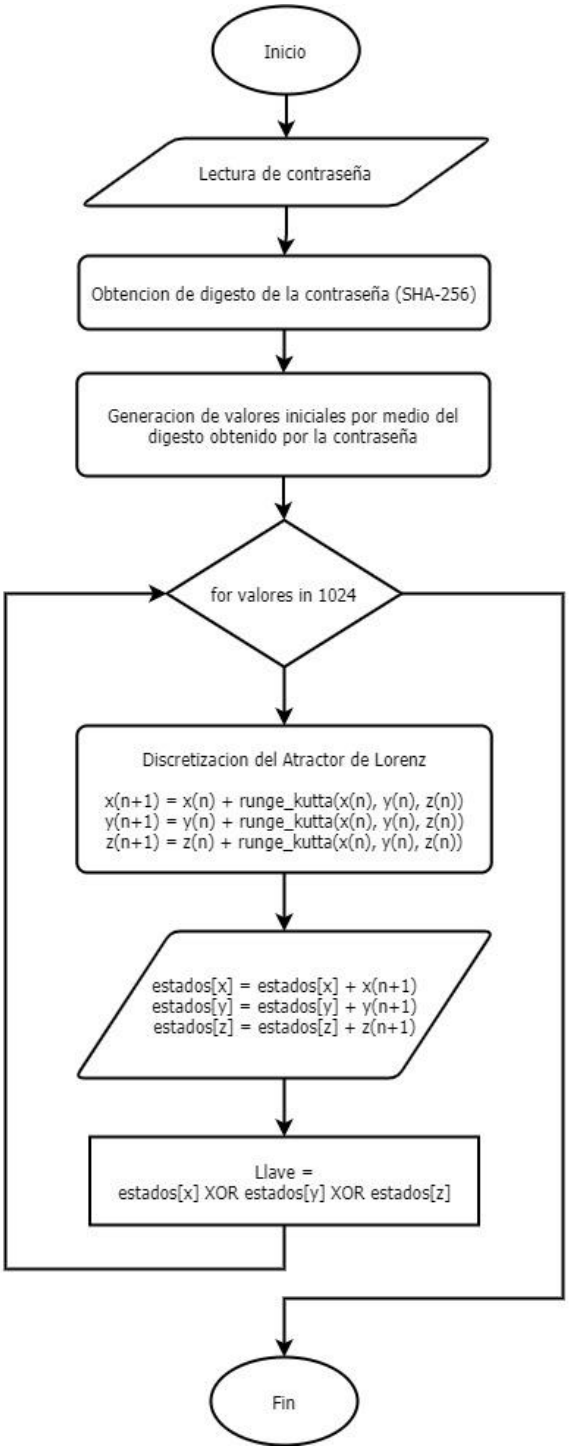


Figura 3.6 *Diagrama de flujo de la generación de la llave privada. Consistente en iterar el algoritmo de Runge-Kutta 1024 veces. La llave se genera a través de la operación XOR entre los resultados obtenidos por el algoritmo de Runge-Kutta obteniendo una llave de 1024 bytes. Fuente: Elaboración propia.*

3.8 Algoritmo criptográfico basado en valores caóticos.

Una vez se obtuvo la llave mediante la generación del atractor de Lorenz se procedió a realizar el algoritmo de cifrado y descifrado digital.

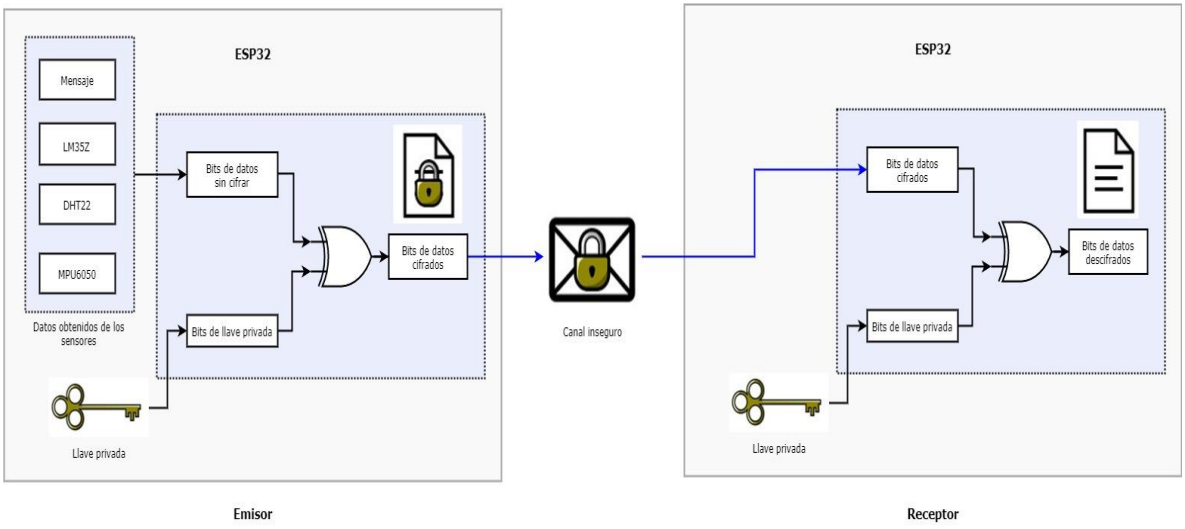


Figura 3.7 *Algoritmo de cifrado simétrico implementado en el microcontrolador ESP32. Fuente: Elaboración propia.*

En la **Figura 3.7** se muestra el diagrama a bloques del algoritmo de cifrado implementado. Cada byte de la información sin cifrar fue puesto en la operación XOR con los bytes de la llave, dando como resultado los bytes cifrados. Mismos que fueron enviados por medio del protocolo de comunicación UART al receptor. El receptor generó su propia llave privada y descifró los bytes aplicando la misma operación XOR a cada byte de la llave y de los datos cifrados, dando como resultado los datos originales. Esta compuerta digital tiene la particularidad de ser reversible en su información, lo que facilitó el proceso criptográfico.

4 RESULTADOS Y DISCUSIÓN.

4.1 Esquemático del diagrama electrónico

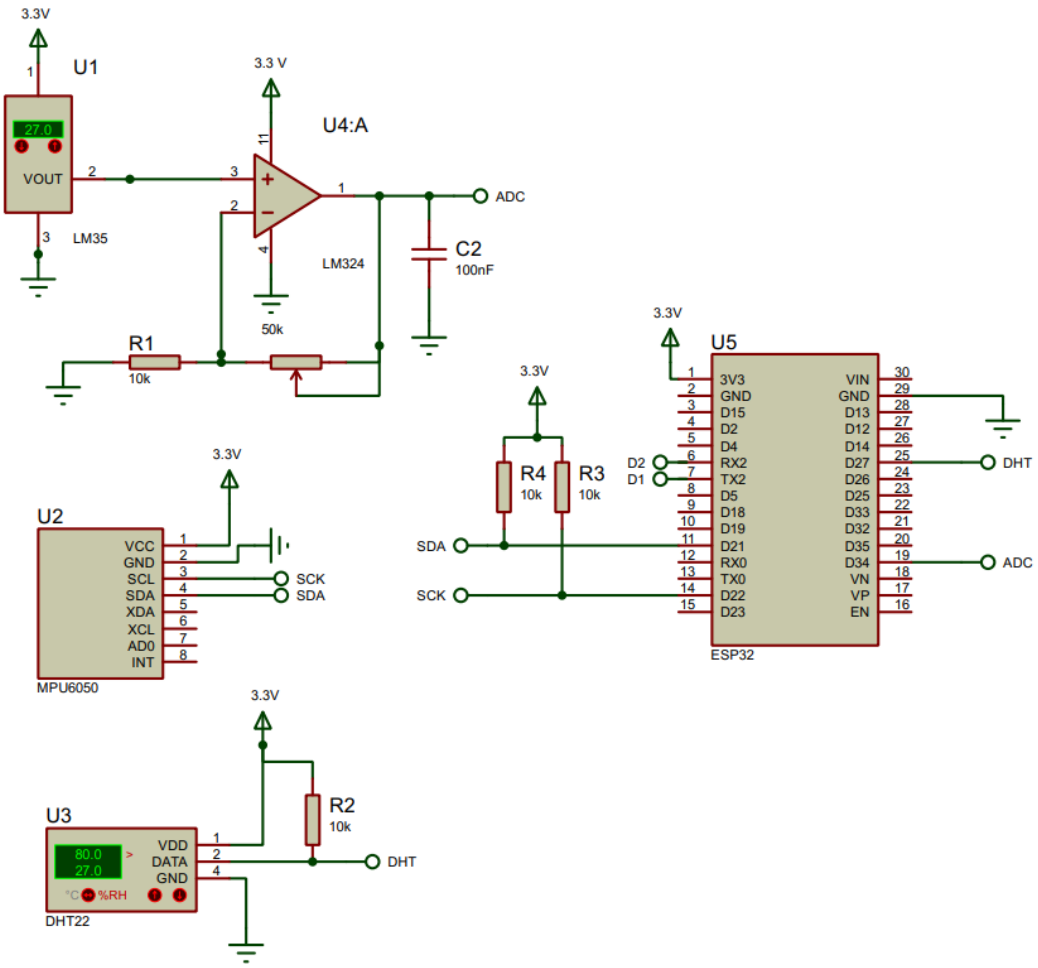


Figura 4.1. *Diseño esquemático del emisor. Enfocado en la conexión entre el microcontrolador ESP32 y los sensores LM35Z, MPU6050 y DHT22. Asimismo, la implementación del LM324 en configuración de amplificador no inversor para el tratamiento de la señal analógica del sensor LM35Z. Fuente: Elaboración propia.*

Las conexiones realizadas en el emisor se pueden verificar en la **Figura 4.1**, es de especial interes el tratamiento de la señal del sensor LM35Z para lo cual se requirió de acondicionar la señal de salida del sensor, por medio del amplificador operacional LM324 en configuración de amplificador no inversor. Cumpliendo con la función de salida expresada en la **Ecuacion (3.3)**

$$ADC = V_{LM35}\left(1+\frac{10k}{10k}\right) = 2V_{LM35}\,[V] \tag{3.3}$$

Por parte del receptor, se establecio la comunicacion mediante en protocolo *UART* entre emisor y receptor. Se puede observar esta conexión en las señales denomiadas *D1* y *D2* de la **Figura 4.2**

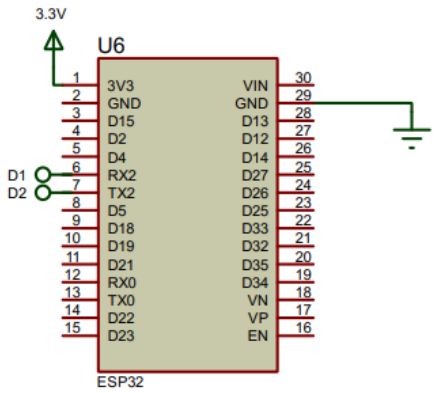


Figura 4.2 *Diseño esquemático del receptor, con el protocolo UART habilitado en los pines RX2 y TX2. Fuente: Elaboración propia*

Finalmente, el prototipo general desarrollado se puede observar en la **Figura 4.3** donde receptor y emisor fueron comunicados mediante el protocolo UART.

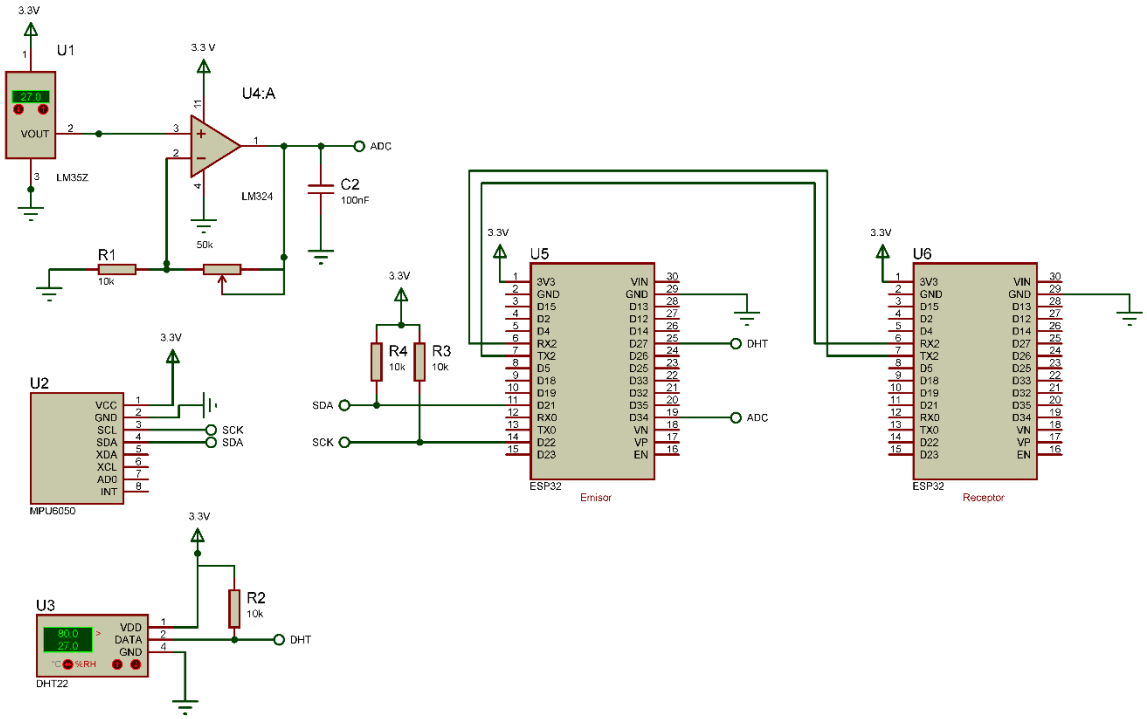


Figura 4.3 *Diagrama esquemático general del emisor y receptor. Fuente: Elaboración propia*

4.2 Prototipo físico.

En la **Figura 4.4** se muestra el prototipo del sistema embebido implementado de forma física mediante dos microcontroladores ESP32. Para asegurar el correcto funcionamiento de los protocolos de comunicación, ambos microcontroladores compartieron el mismo bus de referencia a 0V DC (Tierra). Los sensores fueron configurados para trabajar a un riel de voltaje de 3.3V DC en corriente directa, misma que fue suministrada por el regulador de voltaje con el que cuenta la placa de desarrollo.

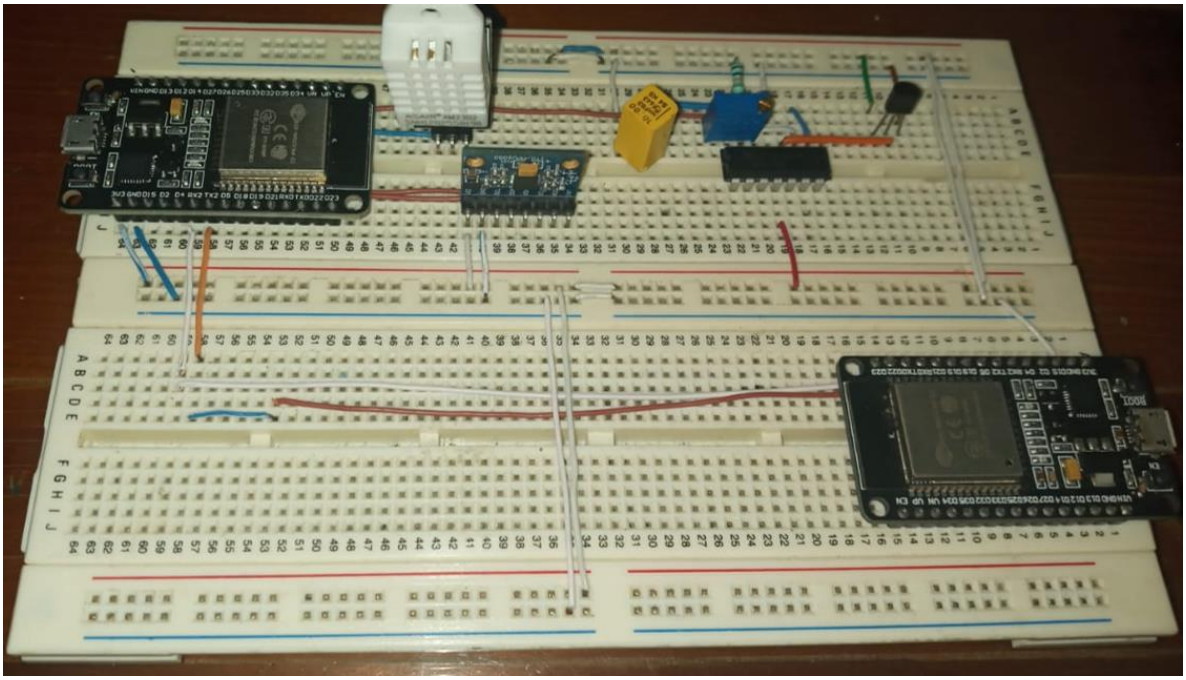


Figura 4.4 *Prototipo desarrollado físicamente en protoboard. Se muestra tanto el emisor en la parte superior, como el receptor en la parte inferior. Fuente: Elaboración propia*

4.3 Resultados del modelo discreto de Lorenz.

Al ejecutar el algoritmo, se logró almacenar los datos obtenidos por la discretización del atractor de Lorenz por medio del microcontrolador. Los cuales fueron analizados mediante software, obteniendo los siguientes resultados.

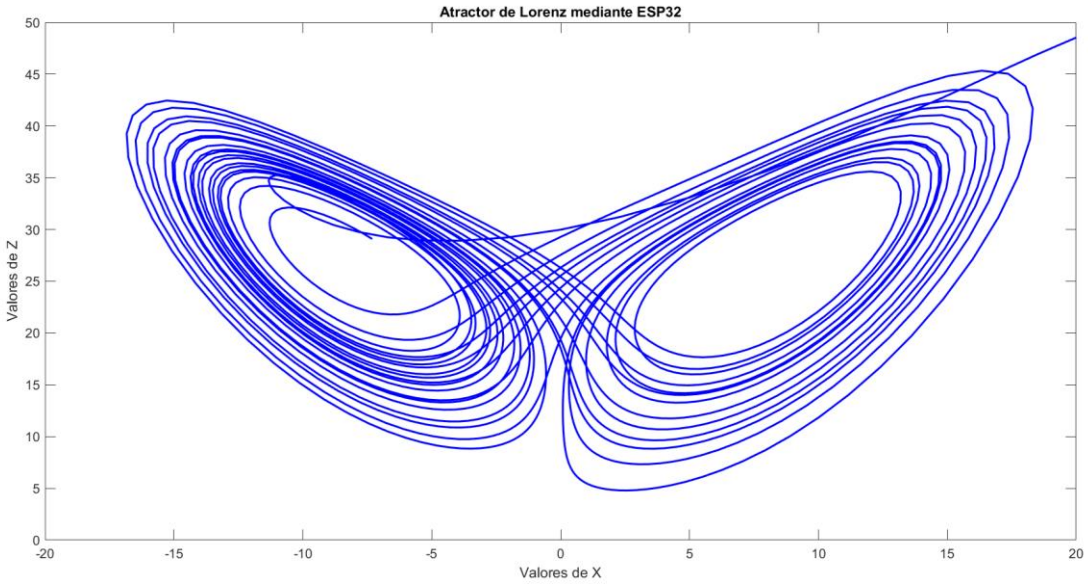


Figura 4.5 *Atractor de Lorenz generado por el ESP32 mediante la digitalización del sistema. Fuente: Elaboración propia*

La **Figura 4.5** representa el atractor generado por medio de las instancias expresadas en los capítulos previos. Es fácil notar que los datos generados tienden a formar la clásica mariposa del atractor de Lorenz, con lo cual se puede asegurar la estabilidad del sistema y la repetibilidad del experimento.

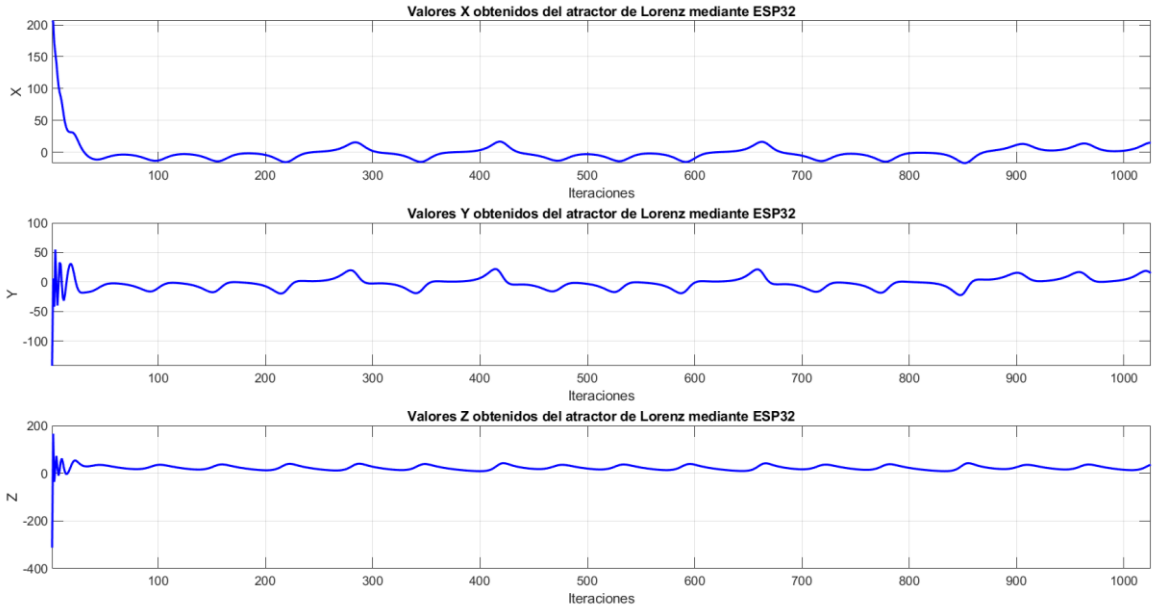


Figura 4.6 *Resultados del atractor de Lorenz en función al número de iteración generada por el ESP32 y el valor de la variable X, Y, Z de forma descendente. Fuente: Elaboración propia*

Por otro lado, el análisis de las variables en función al número de iteraciones realizadas se muestra en la **Figura 4.6**, se concluye que los valores obtenidos cumplen con el requisito de pseudo aleatoriedad y aperiodicidad, lo que impide al atacante utilizar herramientas de criptoanálisis para romper el cifrado.

4.4 Respuesta del atractor de Lorenz en Salida analógica del convertidor DAC.

Se implemento el atractor de Lorenz a través de la salida del *Digital to Analogic Converter* (DAC) con el que cuenta el microcontrolador ESP32. Los resultados tanto analógicos como digitales fueron considerados verificados.

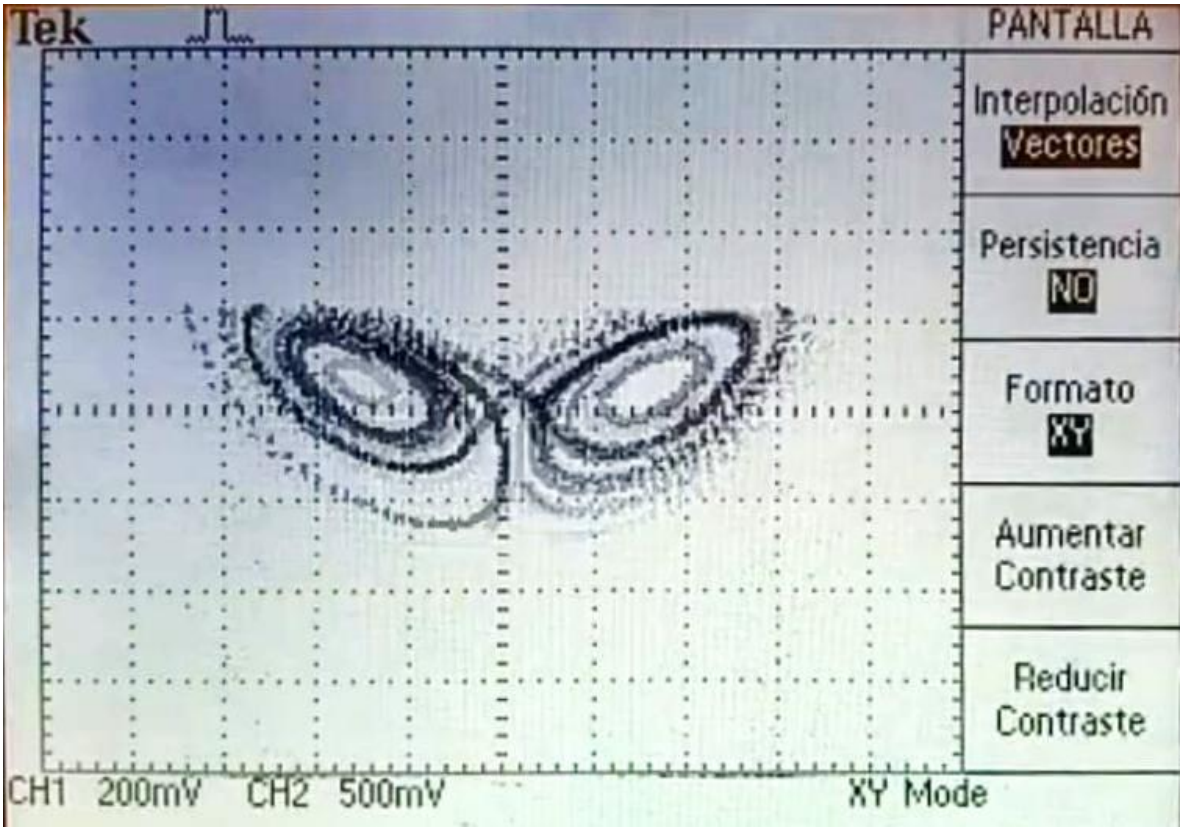


Figura 4.7 *Atractor de Lorenz obtenido por las salidas del Convertidor Digital a Analógico (DAC) del ESP32. Comprobando la naturaleza del atractor.*
Fuente: Elaboración propia.

En la **Figura 4.7** se muestra el resultado tras la visualización en el osciloscopio para las señales analógicas correspondientes a X y Z del atractor de Lorenz. Es fácil observar que se cumple con las mismas características establecidas con anterioridad para el atractor de Lorenz.

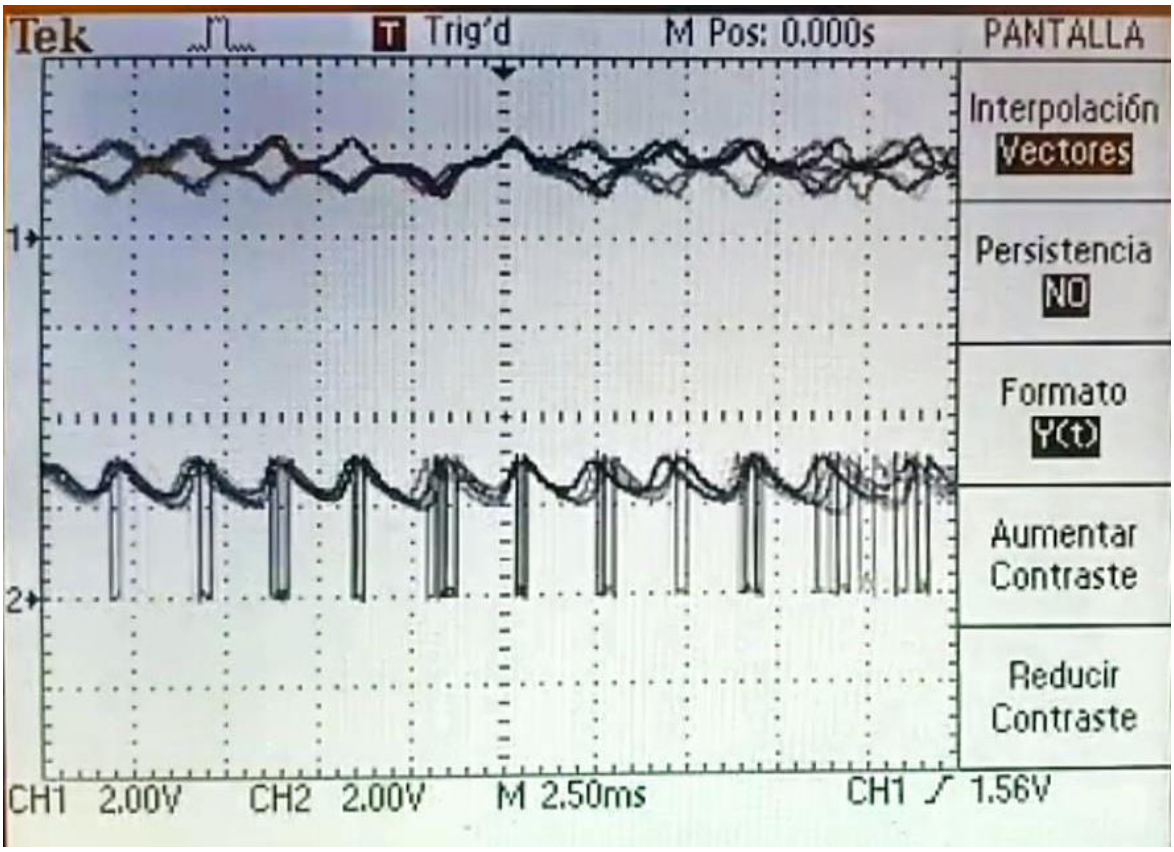


Figura 4.8 *Variables X, Z del Atractor de Lorenz en función al tiempo. Obtenidas por las salidas del convertidor digital a analógico del ESP32. Fuente: Elaboración propia*

Por su parte, el análisis de los resultados del atractor de Lorenz en función al tiempo se puede observar en la **Figura 4.8**, al realizar una comparación con los datos gráficos de forma digital en la **Figura 4.6** se puede asegurar que los valores obtuvieron el resultado esperado, y nuevamente se asegura la no repetibilidad de los datos y la existencia de la señal aperiódica.

4.5 Resultados de la lectura y cifrado de datos por parte del emisor.

Los datos obtenidos de los sensores fueron presentados en pantalla con el objetivo de verificar la correcta lectura de estos. En la **Figura 4.9** se puede observar una captura del software donde se muestran los datos sin cifrar por parte del emisor. Por lo cual, se puede afirmar que los sensores se leyeron correctamente.

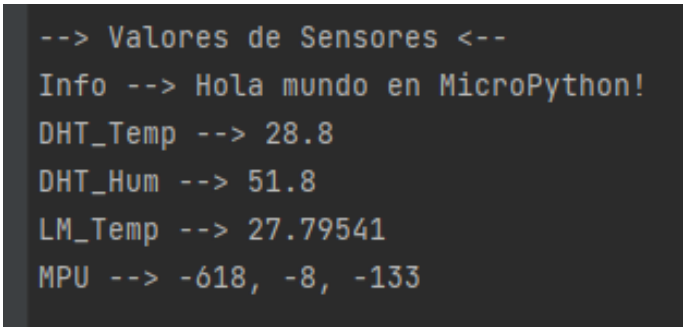


Figura 4.9 *Captura de pantalla del software para la visualización de MicroPython en el ESP32. Se muestran los valores de los sensores obtenidos por el microcontrolador. Fuente: Elaboración propia.*

En la **Figura 4.10** se muestra el resultado del sistema de cifrado, el cual es visualizado como una serie de números enteros, que fueron enviados por el protocolo UART. Cabe mencionar que las etiquetas son meramente informativas, el cifrador trabaja únicamente con las cadenas de números.

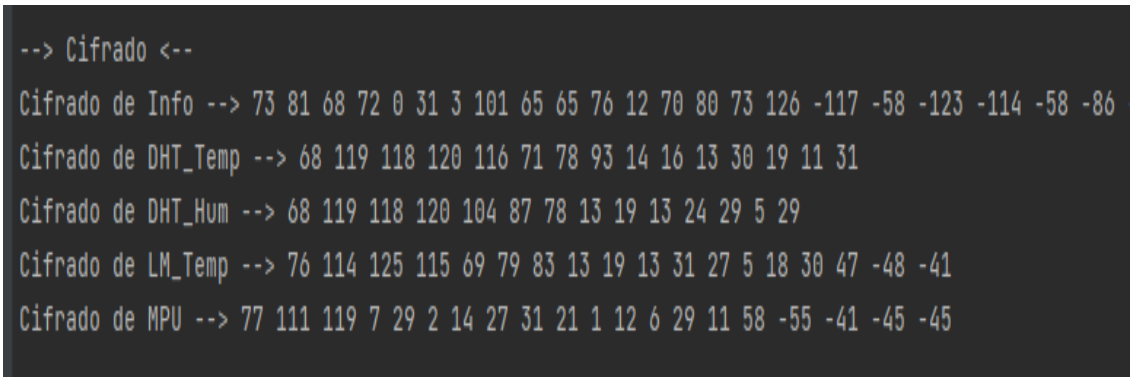


Figura 4.10 *Captura de pantalla del software para la visualización de MicroPython en el ESP32. Se muestran los valores cifrados por el ESP32, mismos que fueron enviados al receptor. Fuente: Elaboración propia.*

4.6 Resultados de descifrado por parte del receptor.

Una vez el emisor envía la información cifrada, el receptor lee dicha información y la muestra en pantalla, para una mejor lectura, se agregaron etiquetas a los datos. En la **Figura 4.11** se muestra la información leída y descifrada por el receptor, al realizar una comparación con la información enviada por el emisor, en la **Figura 4.10** se afirma el correcto funcionamiento del algoritmo de descifrado.

Dado que el sistema se encuentra trabajando en tiempo real, las medidas de los sensores cambian en función al tiempo, por lo cual se estableció el mensaje fijo (Info) para poder tener un punto de comparación más acertado en cuanto al cifrado y descifrado de los datos.

Figura 4.11 *Captura de pantalla del software para la visualización de MicroPython en el ESP32. Se muestra la correcta recepción y descifrado de la información por parte del receptor. Fuente: Elaboración propia.*

4.7 Simulación de ataque de intercepción.

A pesar de la verificación del correcto funcionamiento del sistema para cifrar y descifrar información, fue imprescindible comprobar la confidencialidad de algoritmo, para lo cual se diseñó un ataque de intercepción, por medio de una tercera entidad a la que se le denomino atacante.

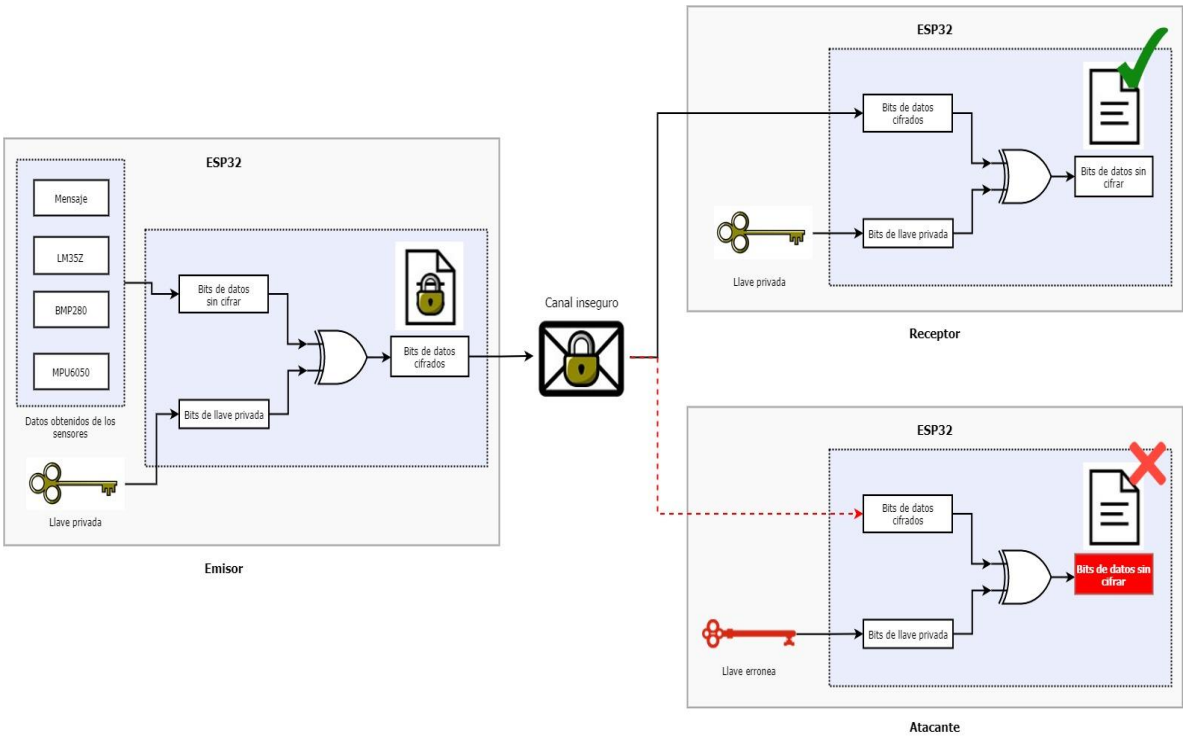


Figura 4.12 *Diagrama a bloques de la simulación de ataque de intercepción para la verificación de la seguridad del sistema. Fuente: Elaboración propia.*

La simulación se llevó a cabo siguiendo el diagrama a bloques presentado en la **Figura 4.12**. En la cual el atacante obtiene los datos cifrados por medio del canal inseguro. Sin embargo, este cuenta con una contraseña diferente a la original. El cambio implementado corresponde a 1 bit de información en la contraseña.

4.8 Resultados del ataque.

Durante la simulación se ejecutó el sistema de forma tal que el emisor y receptor funcionaran de forma ideal, compartiendo la contraseña: *ElectroVigia2021!!* Mientras que al atacante se le otorgo la capacidad de interceptar los datos enviados por el emisor e intento descifrar la información mediante la contraseña: *ElectroVigia2020!!*

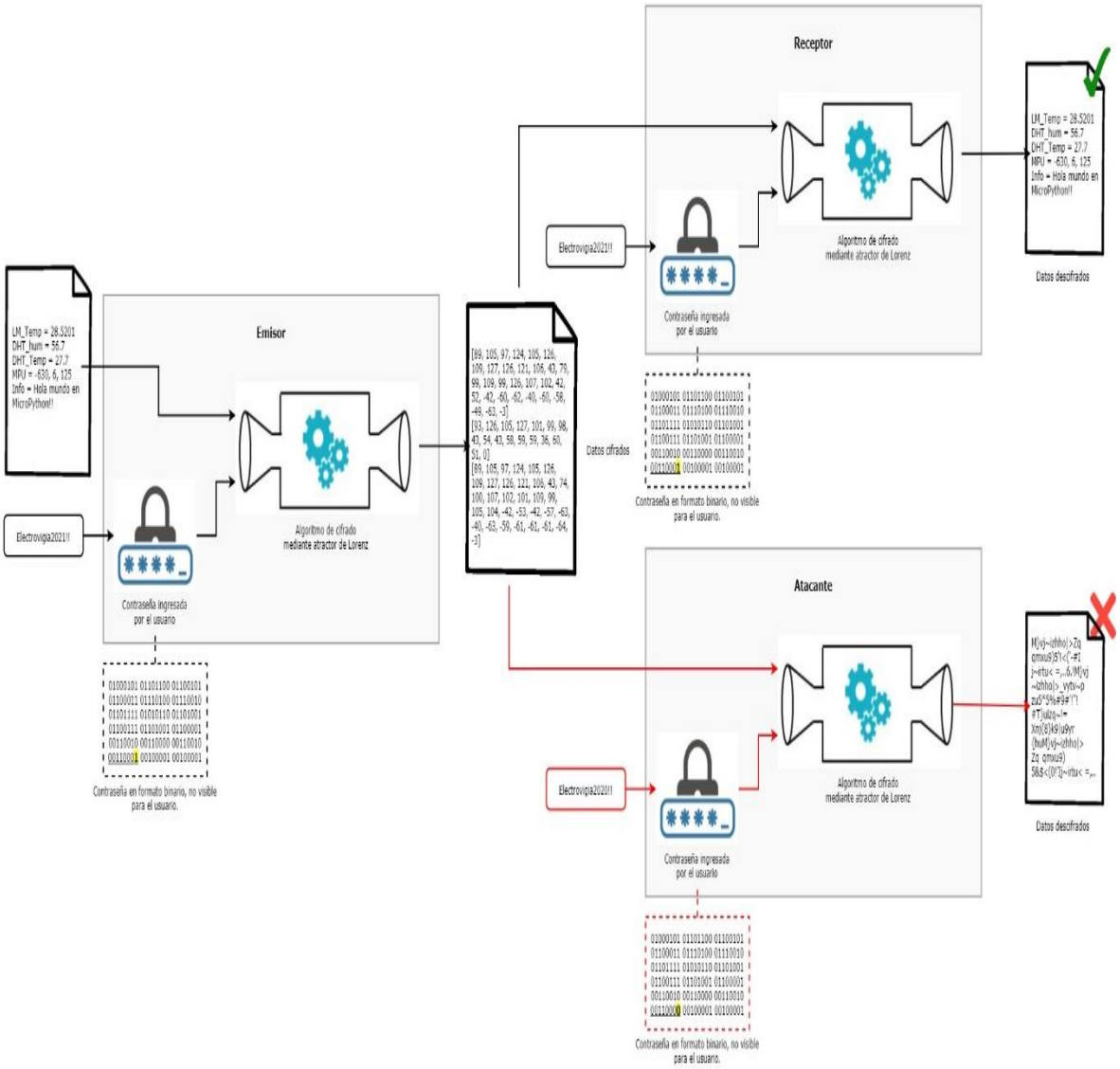


Figura 4.13 *Diagrama a bloques de los resultados obtenidos en la simulación de ataque al sistema. Es posible observar que los resultados por el atacante son ilegibles. Fuente: Elaboración propia.*

La metodología implementada, así como resultados obtenidos se presentan en la **Figura 4.13**. La contraseña fue ingresada en formato ASCII, sin embargo, el sistema la

determina en su forma binaria, dentro de la misma figura se puede observar dicha forma, donde se encuentra señalado de color amarillo el bit que se modificó.

A pesar de que el atacante y el receptor comparten el mismo sistema de descifrado, para el atacante no fue posible recuperar la información original dado el cambio realizado en la contraseña. A continuación, se describen los resultados obtenidos durante el proceso de simulación de ataque. La información original sin cifrar enviada por el emisor fue:

```
Info = Hola mundo en MicroPython!
DHT_Temp = 28.8
DHT_Hum = 51.8
LM_Temp = 27.79541
MPU = -618, -8, -133
```

En la **Tabla 4.1** se muestran los resultados obtenidos por el atacante y el receptor. Se verifica que ambas entidades contaban con los mismos datos iniciales. Ante la modificación de un bit de la contraseña, se provocó que el resultado obtenido por el atacante sea ilegible para el ser humano.

Tabla 4.1 *Resultados obtenidos en la simulación de ataque, mediante el cambio de un bit en la contraseña establecida por el receptor y el atacante. Demostrando la confidencialidad del sistema. Fuente: Elaboración propia.*

	Receptor	**Atacante**
Datos cifrados recibidos o interceptados.	73 81 68 72 0 31 3 101 65 65 76 12 70 80 73 126 -117 -58 -123 -114 -58 -86 -114 -123 -101 121 70 108 97 124 124 125 51 68 119 118 120 116 71 78 93 14 16 13 30 19 11 31 68 119 118 120 104 87 78 13 19 13 24 29 5 29 76 114 125 115 69 79 83 13 19 13 31 27 5 18 30 47 -48 -41 77 111 119 7 29 2 14 27 31 21 1 12 6 29 11 58 -55 -41 -45 -45	73 81 68 72 0 31 3 101 65 65 76 12 70 80 73 126 -117 -58 -123 -114 -58 -86 -114 -123 -101 121 70 108 97 124 124 125 51 68 119 118 120 116 71 78 93 14 16 13 30 19 11 31 68 119 118 120 104 87 78 13 19 13 24 29 5 29 76 114 125 115 69 79 83 13 19 13 31 27 5 18 30 47 -48 -41 77 111 119 7 29 2 14 27 31 21 1 12 6 29 11 58 -55 -41 -45 -45
Contraseña en formato ASCII	ElectroVigia2021!!	ElectroVigia2020!!
Contraseña en formato binario	01000101 01101100 01100101 01100011 01110100 01110010 01101111 01010110 01101001 01100111 01101001 01100001 00110010 00110000 00110010 00110001 00100001 00100001	01000101 01101100 01100101 01100011 01110100 01110010 01101111 01010110 01101001 01100111 01101001 01100001 00110010 00110000 00110010 00110000 00100001 00100001

Datos descifrados	Info = Hola mundo en MicroPython! DHT_Temp = 28.8 DHT_Hum = 51.8 LM_Temp = 27.79541 MPU = -618, -8, -133	M}vj~izhho|>Zq• qmxu9)5'!<('- #j~irt!""T}uiq~!=Xnj{8}k9|u9yr {huM}vj~izhho|>Zq• qmxu9)5' %<-%(.j~irtu (&#$#%T}uizqZq• qmxu9)5&# <"0%$- j~irtu<}uizq~!=Xnj{8}k9|u9yr{ huM}vj~izhho|>Zq)5'!<.$-#- Ij~irtu<$'0))M}vj~izhho|>_vytv ~pzu5*5%-9!"(-- T}uizq~!=Xnj{8k9|u9yr{hu

Los resultados visualizados por parte del atacante se muestran en la **Figura 4.14**. Como se observa los resultados descifrados por el atacante son ilegibles. Con lo cual se aseguró la confidencialidad de los datos, dando como resultado la robustez del sistema, al ser altamente susceptible a pequeños cambios.

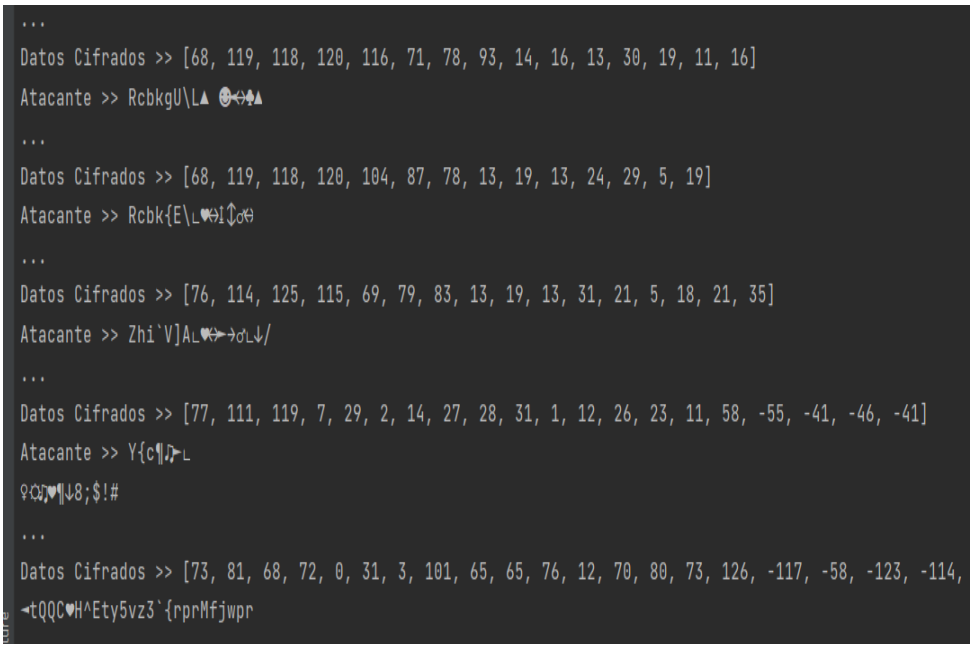


Figura 4.14 *Captura de pantalla del software para la visualización de MicroPython en el ESP32. Lectura por parte del atacante y muestra del correcto funcionamiento del cifrador al no permitir que el atacante pueda leer la información. Fuente: Elaboración propia.*

4.9 Comparación de datos entre atacante y receptor.

Tras realizar el análisis funcionar del sistema, se procedió a realizar una comparación numérica de los resultados obtenidos durante el ataque.

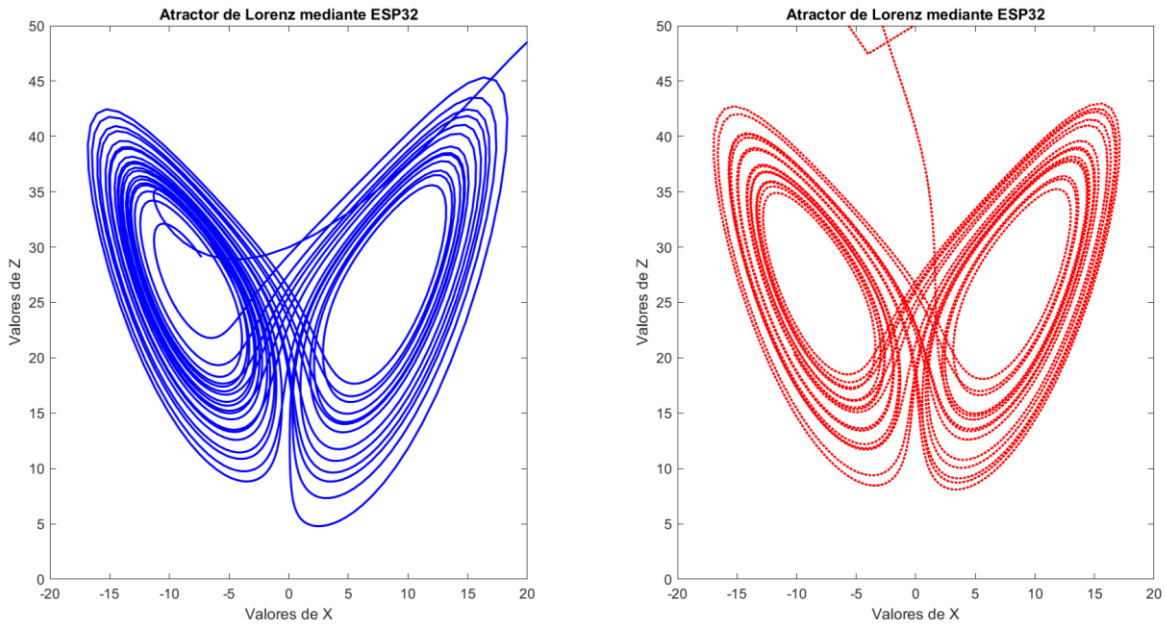


Figura 4.15 *Resultado del atractor de Lorenz generado por el receptor (Azul) y el atacante (Rojo). Fuente: Elaboración propia.*

En la **Figura 4.15** se observan al atractor generado por el receptor (Azul) y el del atacante (Rojo), ambas entidades cuentan con un atractor estable y semejante a pesar de contar con condiciones iniciales diferentes. Lo cual se atribuyó a la naturaleza del atractor de Lorenz.

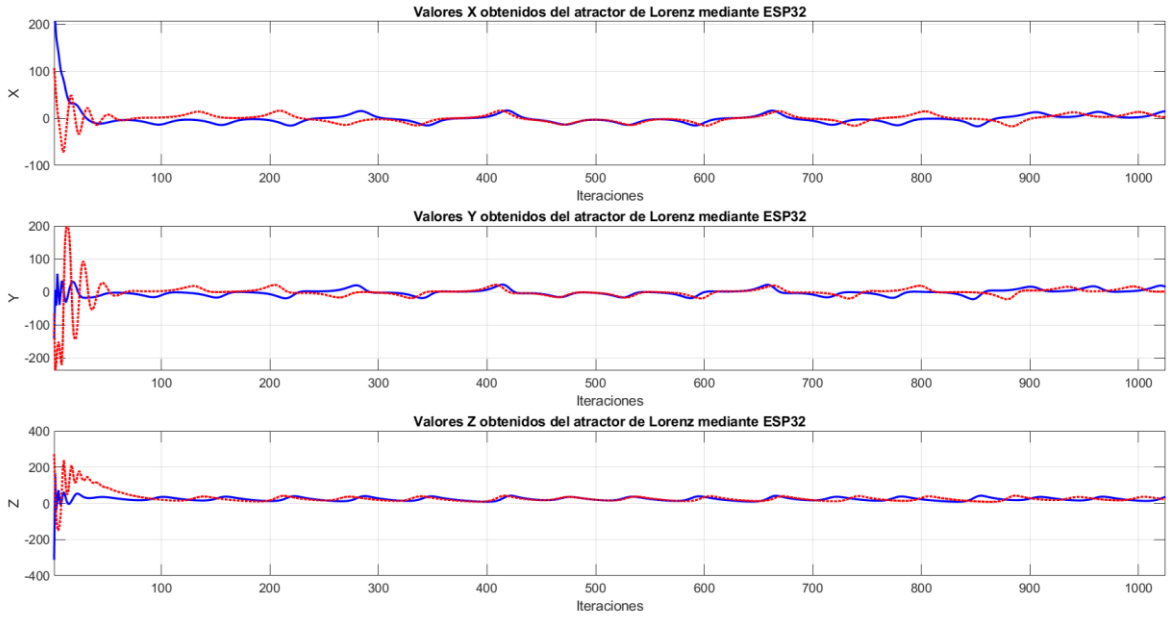


Figura 4.16 *Resultados de las variables X, Y, Z generadas por el receptor (Azul) y el atacante (Rojo). Demostrando la confidencialidad del sistema en función a la no repetibilidad por parte del atacante. Fuente: Elaboración propia*

A pesar de que los atractores mostrados en la **Figura 4.15** resultaran semejantes, en la **Figura 4.16** ser demostraron las diferencias entre ambos atractores, en función a sus variables. Aunque ambos atractores tienden a seguir un comportamiento semejante, los valores obtenidos por cada atractor son sutilmente diferentes, lo cual aseguro la generación de una llave diferente para ambas entidades. Finalmente se comprobó la seguridad, confidencialidad, robustez y sensibilidad del sistema implementado.

4.10 Comparación con modelos semejantes.

Finalmente, obtuvo la comparación del sistema desarrollado con algoritmos ya establecidos y discutidos en el **Capítulo 2 Antecedentes**.

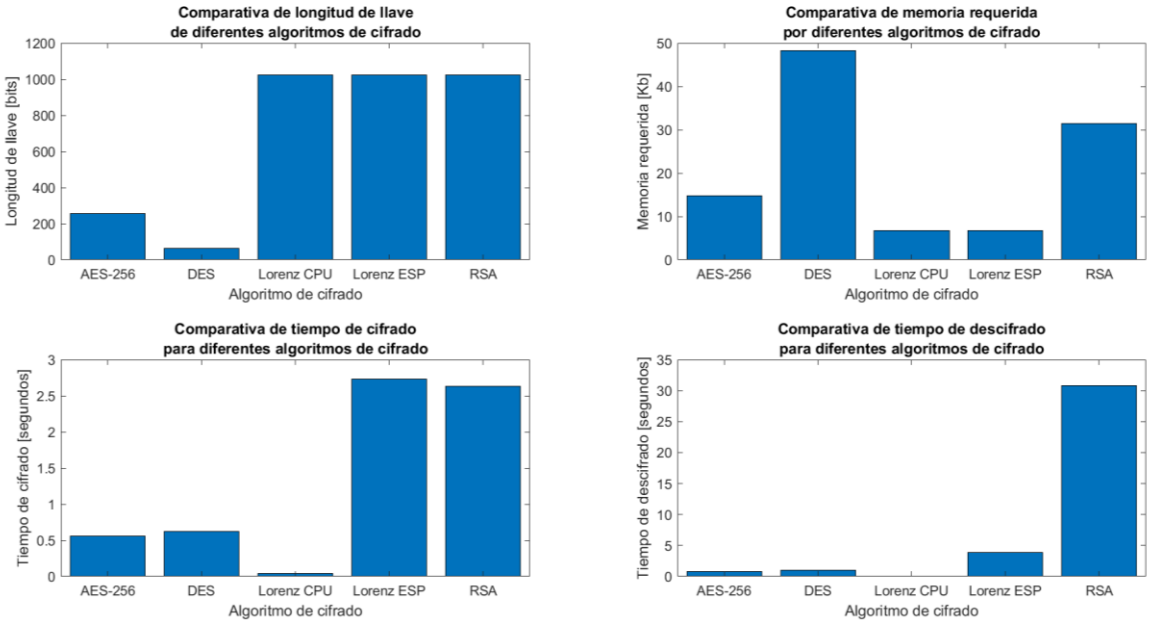


Figura 4.17 *Resultados obtenidos mediante el cifrador desarrollado implementado en el ESP32 (Lorenz ESP) y un equipo de cómputo (Lorenz CPU) comparados con algoritmos criptográficos. Demostrando las ventajas del cifrador caótico. Fuente: Elaboración propia.*

En la **Figura 4.17** se muestra de forma gráfica las diferentes características de los algoritmos. En cuanto al tamaño de la llave, el sistema desarrollado obtuvo un tamaño de llave igual a algoritmo RSA, lo que lo deja por arriba de otros sistemas como AES y DES. Una de las áreas donde se obtuvo una gran ventaja sobre otros algoritmos fue en el espacio de memoria requerido. El sistema se desarrolló tomando en cuenta las limitaciones de

memoria presentes en los sistemas embebidos, teniendo éxito en este punto al tener un tamaño menor a 10 Kb.

En cuanto a la velocidad de cifrado y descifrado, el sistema se probó en el ESP32 a una velocidad de reloj 240 MHz, dando como resultado tiempos menores a los 5 segundos. Si bien los otros algoritmos son más rápidos, no se debe de perder de vista que estos son implementados en equipos con capacidades de cómputo mayores.

Se realizo una prueba de cifrado y descifrado con ayuda del equipo de cómputo descrito en **Materiales y métodos.** Con lo cual se obtuvieron tiempos menores a los otros algoritmos. Dicha información se resume en la **Tabla 4.2**.

Tabla 4.2 *Comparación de algoritmos de cifrado con el sistema criptográfico desarrollado mediante el Atractor de Lorenz. Fuente: Elaboración propia*

	Longitud de llave [Bytes]	**Memoria requerida [Kb]**	**Tiempo de Cifrado [s]**	**Tiempo de descifrado [s]**
AES-256	256	14.7	0.562	0.815
DES	64	47.2	0.620	0.887
RSA	1024	31.5	2.636	30.779
Lorenz ESP32	1024	6.8	2.732	3.831
Lorenz CPU	1024	6.8	0.0399	0.04201

Cabe recalcar que el algoritmo criptográfico se verifico en un equipo de cómputo (Lorenz CPU) para asegurar la multiplataforma del sistema. Los resultados fueron favorables como se muestra en la **Tabla 4.2**.

5 CONCLUSIONES

Como se mencionó en **Antecedentes** los actuales algoritmos criptográficos están diseñados para trabajar sobre equipos con una capacidad de cómputo alta; en comparación con los sistemas embebidos. Al concluir el diseño, construcción y programación del sistema electrónico de bajo costo, para el cifrado de una señal digital, basado en la teoría de sistemas dinámicos caóticos se verificó el correcto funcionamiento del cifrador. Lo anterior se comprobó y verifico en **Resultados y discusión**..

Se demostró la factibilidad de los sistemas dinámicos estocásticos como generadores de números pseudoaleatorios al ser considerados como señales aperiódicas. La alta sensibilidad a los cambios en las condiciones iniciales fue recurso fundamental para asegurar la confidencialidad del algoritmo. Lo anterior se sustentó gracias a la simulación de ataque realizada en el capítulo previo. Cambiar un solo bit de una cadena de 32 bytes es suficiente para impedir que alguna entidad no autorizada pueda descifrar la información protegida.

Por su parte el diseño del algoritmo con uso de la programación orientada a objetos permitió al sistema tener la capacidad de ser ejecutado por dispositivos de baja capacidad de cómputo. Con microcontrolador ESP32 se obtuvo una buena respuesta, sin embargo, el sistema de cifrado (**Código de la clase LorenzAttractor**) fue diseñado para ser multiplataforma y por ende cualquier sistema que soporte Python o MicroPython puede ejecutar el algoritmo criptográfico.

Gracias a que el sistema fue aplicado a protocolos de comunicación estandarizados como I2C, 1-Wire, SPI y UART es posible cambiar los métodos de comunicación del sistema embebido por cualquiera de los estándares antes mencionados. Por ejemplo, el sistema es fácilmente adaptable a las comunicaciones RF por medio del Módulo Lora E32 a través del protocolo UART y I2C.

5.1 Recomendaciones

Al finalizar los experimentos realizados en el presente proyecto de tesis, se observan varios puntos en los cuales el sistema podría mejorar, a continuación, se describe parte de ellos.

Dado que el sistema trabaja en tiempo real y la discretización del atractor de Lorenz permite obtener valores continuamente si el sistema se encuentra computando, es posible simular un cifrador One Time Pad (Deng y Long 2004) considerado uno de los sistemas de cifrado con mayor fortaleza, en la actualidad.

De igual forma, es posible cifrar byte por byte de la información tan pronto se vaya obteniendo el valor del atractor, lo que permitiría no almacenar la llave dentro de la memoria RAM, mejorando la eficiencia del sistema en cuanto a requerimientos de memoria.

Es posible mejorar el sistema, haciendo uso de las características intrínsecas que ofrece MicroPython, optimizar el sistema para obtener la máxima capacidad de procesamiento por medio del desarrollo de hilos y de multiprocesos.

Es indispensable mantener el sistema en continua actualización, lo que disminuirá la posibilidad de ataques, para lo cual se propone cambiar el algoritmo de digesto SHA-256 por el actual algoritmo SHA-512.

Finalmente, se propone optimizar el código desarrollado empleando los nuevos métodos que las versiones actuales de MicroPython ofrecen. Por ejemplo, cambiar los ciclos de iteración ***For*** por la compresión de listas, de esta forma se optimiza el tiempo de ejecución y el tamaño del algoritmo.

6 BIBLIOGRAFÍA

Abood, Omar, y Shawkat Guirguis. 2018. "A Survey on Cryptography Algorithms". *International Journal of Scientific and Research Publications* 8 (julio): 495–516. https://doi.org/10.29322/IJSRP.8.7.2018.p7978.

Al-Hazaimeh, Obaida M., Mohammad F. Al-Jamal, Nouh Alhindawi, y Abedalkareem Omari. 2019. "Image Encryption Algorithm Based on Lorenz Chaotic Map with Dynamic Secret Keys". *Neural Computing and Applications* 31 (7): 2395–2405. https://doi.org/10.1007/s00521-017-3195-1.

Aumasson, Jean-Philippe. 2017. *Serious Cryptography: A Practical Introduction to Modern Encryption*. San Francisco, USA: No Starch Press.

Boneh, Dan, y Victor Shoup. 2015. *A Graduate Course in Applied Cryptography*. London, England: Stanford. https://crypto.stanford.edu/~dabo/cryptobook/draft_0_2.pdf.

Deng, Fu-Guo, y Gui Lu Long. 2004. "Secure direct communication with a quantum one-time pad". *Physical Review A* 69 (5): 052319. https://doi.org/10.1103/PhysRevA.69.052319.

Ghys, Étienne. 2013. "The Lorenz Attractor, a Paradigm for Chaos". En *Chaos: Poincaré Seminar 2010*, editado por Bertrand Duplantier, Stéphane Nonnenmacher, y Vincent Rivasseau, 1–54. Progress in Mathematical Physics. Basel: Springer. https://doi.org/10.1007/978-3-0348-0697-8_1.

Gowda, Shreyank N. 2016. "Innovative enhancement of the Caesar cipher algorithm for cryptography". En *2016 2nd International Conference on Advances in Computing, Communication, Automation (ICACCA) (Fall)*, 1–4. https://doi.org/10.1109/ICACCAF.2016.7749010.

Hercigonja, Zoran. 2016. "Comparative Analysis of Cryptographic Algorithms". *International Journal of Digital Technology & Economy* 1 (2): 127–34.

Lu, Yang, y Li Da Xu. 2019. “Internet of Things (IoT) Cybersecurity Research: A Review of Current Research Topics”. *IEEE Internet of Things Journal* 6 (2): 2103–15. https://doi.org/10.1109/JIOT.2018.2869847.

Morón, José. 2020. *Señales y sistemas*. Sultana del Lago, Editores.

Orman, H. 2003. “The Morris worm: a fifteen-year perspective”. *IEEE Security Privacy* 1 (5): 35–43. https://doi.org/10.1109/MSECP.2003.1236233.

Press, William H., Saul A. Teukolsky, William T. Vetterling, y Brian P. Flannery. 2007. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press.

Salomaa, Arto. 2013. *Public-Key Cryptography*. Springer Science & Business Media.

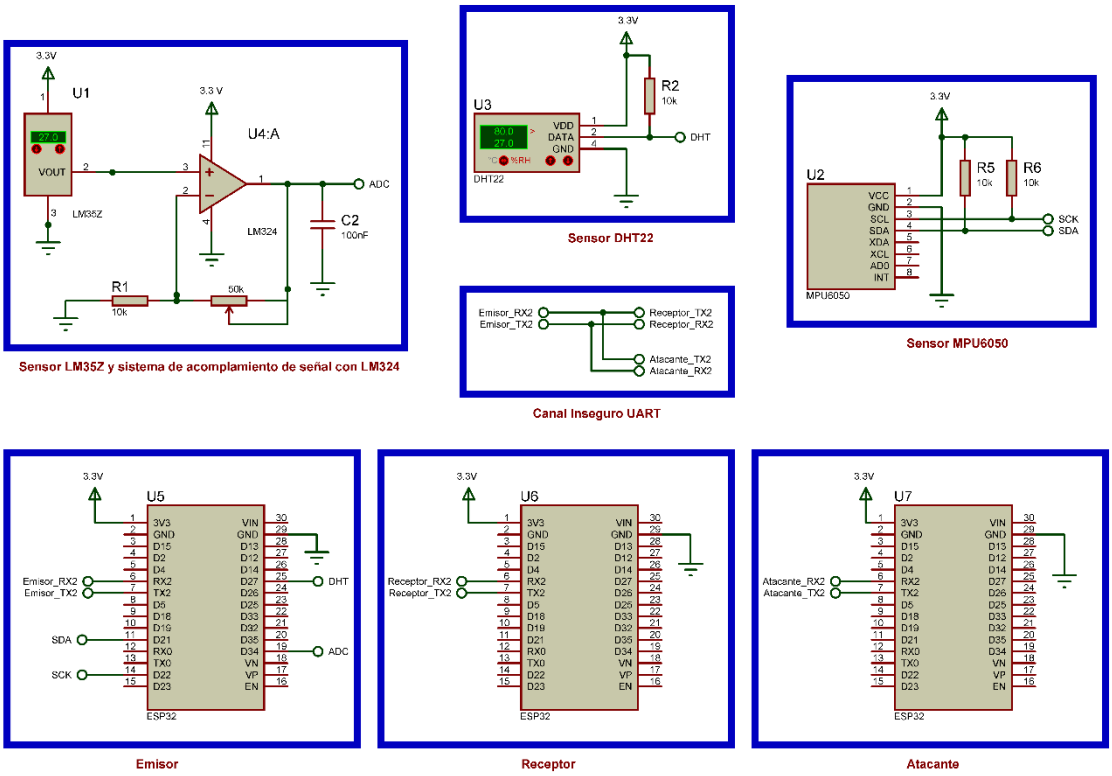
Sánchez, V., E. A. Romero, Manuel Priám Rodríguez, M. Sánchez, J. L. Rojas, y R. González. 1999. “Criptografía de señal: Sincronización de dos osciladores caóticos”. *Mundo electrónico*, núm. 303: 58–60.

Simakov, Nikolay A., Martins D. Innus, Matthew D. Jones, Joseph P. White, Steven M. Gallo, Robert L. DeLeon, y Thomas R. Furlani. 2018. “Effect of Meltdown and Spectre Patches on the Performance of HPC Applications”. *arXiv:1801.04329 [cs]*, enero. http://arxiv.org/abs/1801.04329.

Union Internacional de Telecomunicaciones, ITU. 1991. “X.800 : Arquitectura de seguridad de la interconexión de sistemas abiertos para aplicaciones del CCITT”. Arquitectura de seguridad de la interconexión de sistemas abiertos para aplicaciones del CCITT. el 22 de marzo de 1991. https://www.itu.int/rec/T-REC-X.800-199103-I/es.

7 APÉNDICES

A. Hardware implementado



Apéndice A. *Esquemático final implementado en el prototipo desarrollado.*

Fuente: Elaboración propia.

B. Código main del emisor

```
from Libraries import LorenzAttractor
from Libraries import Embedded
from Libraries import Accel
from Libraries import DHT
import time



def list2str(numbers_list: list) -> str:
    aux = list(map(str, numbers_list))
    res = " ".join(aux)
    return res



def str2list(t1: str) -> list:
    l1 = t1.split()
    l2 = list(map(int, l1))
    return l2



if __name__ == '__main__':
    uc = Embedded()
    entity = LorenzAttractor(secret='ElectroVigia2021!!')
    entity.get_key2(offset=1024)
```

```
# Configuracion de UART
try:
    print(f'Configurando UART...')
    uc.config_UART(number=2, tx=17, rx=16)
    print(f'UART Listo!!!')
except Exception as ints:
    print(ints)
    print('Error en UART')

# Configuracion de I2C
try:
    print(f'Configurando I2C...')
    uc.config_I2C(sda=21, scl=22)
    print(f'I2C Listo!!!')
except Exception as ints:
    print(ints)
    print('Error en I2C')

# Configuracion de GPIO
try:
    print(f'Configurando GPIO...')
    uc.config_pin_out(number=2)
    uc.config_pin_out(number=5)
    uc.config_ADC(number=34)
    print(f'GPIO Listo!!!')
except Exception as ints:
```

```
        print(ints)
        print('Error en GPIO')

    # Configurando Acelerometro
    try:
        print(f'Configurando Acelerometro...')
        accel = accel(uc.i2c)
        print(f'Acelerometro Listo!!!')
    except Exception as ints:
        print(ints)
        print('Error en Acelerometro')

    # Configurando DHT22
    try:
        print(f'Configurando DHT22...')
        temp_hum = DHT(27)
        print(f'DHT22 Listo!!!')
    except Exception as ints:
        print(ints)
        print('Error en DHT22')

    Data = {'LM_Temp': "", 'DHT_Temp': "", 'DHT_Hum': "", 'MPU': "",
            'Info': 'Hola mundo en MicroPython!'}

    while True:
        uc.pin_change_state(number=2)
```

```
        try:
            aux = accel.get_values()
            Giros_values = str(int(aux.get("GyX"))) + ', ' + \
                           str(int(aux.get("GyY"))) + ', ' + \
                           str(int(aux.get("GyZ")))
            Data['MPU'] = str(Giros_values)
        except Exception as ints:
            print(ints)
            print('Error al leer MPU6050')

        try:
            Data['DHT_Hum'] = str(temp_hum.humidity())
            Data['DHT_Temp'] = str(temp_hum.temperature())
        except Exception as ints:
            print(ints)
            print('Error al leer DHT22')

        try:
            Data['LM_Temp'] = str(330 * uc.read_ADC(number=34) / (4096 *
2))
        except Exception as ints:
            print(ints)
            print('Error al leer LM35')

        print("\n--> Valores de Sensores <--")
        for key, values in Data.items():
```

```
            print(f"{key} --> {values}")
            aux = key + ' = ' + values
            data = entity.encrypt_text2(aux)
            encrypt = list2str(data)
            try:
                uc.send_uart(kind='>', value=encrypt)
            except Exception as ints:
                print(ints)
                print(f'Error al enviar {aux}')


        print("\n--> Cifrado <--")
        for key, values in Data.items():
            try:
                aux = key + ' = ' + values
                data = entity.encrypt_text2(aux)
                encrypt = list2str(data)
                print(f"Cifrado de {key} --> {encrypt}")
            except Exception as ints:
                print(ints)
                print(f'Error al cifrar')
```

C. Código main del receptor.

```
from Libraries import LorenzAttractor

from Libraries import Embedded

from Libraries import accel

from Libraries import DHT


import time



def list2str(numbers_list: list) -> str:

    aux = list(map(str, numbers_list))

    res = " ".join(aux)

    return res



def str2list(t1: str) -> list:

    l1 = t1.split()

    l2 = list(map(int, l1))

    return l2



if __name__ == '__main__':

    uc = Embedded()

    uc.config_pin_out(number=2)
```

```
try:
    print(f'Configurando UART...')
    uc.config_UART(number=2, tx=17, rx=16)
    print(f'UART Listo!!!')
except Exception as inst:
    print(inst)
    print('Hubo un error en UART')
    input('Pulsa una tecla para continuar')


entity = LorenzAttractor(secret='ElectroVigia2021!!')
entity.get_key2(offset=1024)


while True:
    uc.pin_change_state(number=2)


    if uc.uart.any():
        rev_bin = uc.uart.read()
        try:
            rev_str = rev_bin.decode()[3:]
            data = str2list(rev_str)
            try:
                info = entity.decrypt_text2(data)
                print(f"Datos Cifrados >> {data}")
                print(f"Datos Descifrados >> {info}")
            except Exception as inst:
                print(inst)
```

```
                print(f"Error al descrifrar {data}")
            except Exception as inst:
                print(inst)
                print(f'Error al convertir {rev_bin}')
            print('...')
```

D. Código main del atacante.

```
import time



def list2str(numbers_list: list) -> str:
    aux = list(map(str, numbers_list))
    res = " ".join(aux)
    return res



def str2list(t1: str) -> list:
    l1 = t1.split()
    l2 = list(map(int, l1))
    return l2



if __name__ == '__main__':
    uc = Embedded()
    uc.config_pin_out(number=2)


    try:
        print(f'Configurando UART...')
        uc.config_UART(number=2, tx=17, rx=16)
        print(f'UART Listo!!!')
```

```
except Exception as inst:
    print(inst)
    print('Hubo un error en UART')
    input('Pulsa una tecla para continuar')

Atacante = LorenzAttractor(secret='ElectroVigia2020!!')
Atacante.get_key2(offset=1024)

while True:
    uc.pin_change_state(number=2)

    if uc.uart.any():
        rev_bin = uc.uart.read()
        try:
            rev_str = rev_bin.decode()[3:]
            data = str2list(rev_str)
            try:
                error = Atacante.decrypt_text2(data)
                print(f"Datos Descifrados >> {info}")
                print(f"Atacante >> {error}")
            except Exception as inst:
                print(inst)
                print(f"Error al descrifrar {data}")
        except Exception as inst:
            print(inst)
            print(f'Error al convertir {rev_bin}')
```

```
            print('...')
```

E. Código de la clase LorenzAttractor

```
class LorenzAttractor:
    def __init__(self, secret: str):
        self.constants = [28.0, 10.0, 8.0 / 3.0]  # [Rho, Sigma, Beta]
        self.states = [0, 0, 0, 0]
        self.key = []
        self.password = secret
        self.h = 0.01

    def fx(self, x, y, z) -> float:
        dxdt = self.constants[1] * (y - x)  # Sigma
        return dxdt

    def fy(self, x, y, z) -> float:
        dydt = x * (self.constants[0] - z) - y  # RHo
        return dydt

    def fz(self, x, y, z) -> float:
        dzdt = x * y - self.constants[2] * z  # Beta
        return dzdt

    def runge_kutta2(self, x, y, z, h) -> list:
        k1x = h * self.fx(x, y, z)
        k1y = h * self.fy(x, y, z)
        k1z = h * self.fz(x, y, z)
```

```
        k2x = h * self.fx(x + k1x / 2, y + k1y / 2, z + k1z / 2)
        k2y = h * self.fy(x + k1x / 2, y + k1y / 2, z + k1z / 2)
        k2z = h * self.fz(x + k1x / 2, y + k1y / 2, z + k1z / 2)

        k3x = h * self.fx(x + k2x / 2, y + k2y / 2, z + k2z / 2)
        k3y = h * self.fy(x + k2x / 2, y + k2y / 2, z + k2z / 2)
        k3z = h * self.fz(x + k2x / 2, y + k2y / 2, z + k2z / 2)

        k4x = h * self.fx(x + k3x, y + k3y, z + k3z)
        k4y = h * self.fy(x + k3x, y + k3y, z + k3z)
        k4z = h * self.fz(x + k3x, y + k3y, z + k3z)

        kx = x + ((k1x + 2 * k2x + 2 * k3x + k4x) / 6)
        ky = y + ((k1y + 2 * k2y + 2 * k3y + k4y) / 6)
        kz = z + ((k1z + 2 * k2z + 2 * k3z + k4z) / 6)



        return [kx, ky, kz, int(kx) ^ int(ky) ^ int(kz)]
        # return [kx, ky, kz, kx * ky * kz]

    def get_key(self, offset):
        self.states = [0, 0, 0, 0]
        self.key = []

        dig = uhashlib.sha256()
```

```
        text = self.password.encode('utf-8')

        dig.update(text)

        digests = dig.digest()


        for byte in digests[0:9]:

            self.states[0] ^= byte

        for byte in digests[10:19]:

            self.states[1] ^= byte

        for byte in digests[20:29]:

            self.states[2] ^= byte


        aux = digests[30] ^ digests[31]


        self.h = 0.015 - (abs(aux / 25555))


        for t in range(1, 810 + offset):

            self.states        =        self.runge_kutta2(self.states[0],
self.states[1], self.states[2], abs(self.h))

            if t > 800:

                self.key.append(int(self.states[3]))


    def encrypt_text(self, aux: str) -> list:


        text = aux.encode('utf-8')


        msg = len(text)
```

```
        self.get_key(offset=msg)
        encrypted = []

        try:
            encrypted = list(map(lambda m, k: int(m) ^ k, text, self.key))
        except (RuntimeError, TypeError, NameError):
            print('Algo salio mal al cifrar')
            print(Exception)
        finally:
            return encrypted

    def decrypt_text(self, text: list) -> str:

        large = len(text)
        self.get_key(offset=large)
        aux = []
        try:
            aux = list(map(lambda k, m: chr(abs(k ^ int(m))), self.key,
text))
        except (RuntimeError, TypeError, NameError):
            print("El mensaje recibido es: ", text)
            print("Algo salio mal al descifrar, revisa la contraseña")
            print(Exception)
        finally:
            msg = "".join(aux)
```

```
            return msg


    def decrypt_text2(self, text: list) -> str:


        large = len(text)
        aux = []
        try:
            aux = list(map(lambda k, m: chr(abs(k ^ int(m))), self.key,
text))
        except (RuntimeError, TypeError, NameError):
            print("El mensaje recibido es: ", text)
            print("Algo salio mal al descifrar, revisa la contraseña")
            print(Exception)
        finally:
            msg = "".join(aux)
            return msg


    def encrypt_text2(self, aux: str) -> list:


        text = aux.encode('utf-8')


        encrypted = []


        try:
            encrypted = list(map(lambda m, k: int(m) ^ k, text, self.key))
        except (RuntimeError, TypeError, NameError):
```

```
            print('Algo salio mal al cifrar')
            print(Exception)
        finally:
            return encrypted

    def get_key2(self, offset):
        self.states = [0, 0, 0, 0]
        self.key = []

        dig = uhashlib.sha256()
        text = self.password.encode('utf-8')
        dig.update(text)
        digests = dig.digest()

        for byte in digests[0:9]:
            self.states[0] ^= byte
        for byte in digests[10:19]:
            self.states[1] ^= byte
        for byte in digests[20:29]:
            self.states[2] ^= byte

        aux = digests[30] ^ digests[31]

        self.h = 0.015 - (abs(aux / 25555))

        for t in range(1, 1024 + offset):
```

```
            self.states        =        self.runge_kutta2(self.states[0],
self.states[1], self.states[2], abs(self.h))

            if t > 800:

                self.key.append(int(self.states[3]))
```

F. Código de la clase Embedded

```
import uhashlib
from machine import UART
from machine import Pin
from machine import ADC
from machine import I2C
import dht
from time import sleep



class Embedded:
    def __init__(self):
        self.pins = {}
        self.adcs = {}
        self.uart = None
        self.i2c = None

    def config_pin_out(self, number: int):
        aux = Pin(number, Pin.OUT)
        aux2 = 'Pin' + str(number)
        self.pins.update({aux2: aux})

    def pin_state(self, number: int, state: int):
        aux2 = 'Pin' + str(number)
        a = self.pins.get(aux2)
```

```
        a.value(state)


    def pin_change_state(self, number: int):
        aux2 = 'Pin' + str(number)
        a = self.pins.get(aux2)
        state = a.value()
        a.value(state ^ 1)


    def config_UART(self, number: int, tx: int, rx: int):
        self.uart = UART(number, tx=tx, rx=rx)
        self.uart.init(baudrate=9600, bits=8, parity=None, stop=1)


    def send_uart(self, kind: str, value: str):
        send = kind + '> ' + value + '\n'
        self.uart.write(send)
        sleep(2)


    def send_uart2(self, value: str):
        send = value + '\n'
        self.uart.write(send)


    def config_ADC(self, number: int):
        adc = ADC(Pin(number))
        adc.atten(ADC.ATTN_11DB)
        aux2 = 'ADC' + str(number)
        self.adcs.update({aux2: adc})
```

```
    def read_ADC(self, number: int) -> int:
        aux2 = 'ADC' + str(number)
        a = self.adcs.get(aux2)
        value = a.read()
        return value


    def config_I2C(self, sda: int, scl: int):
        self.i2c = I2C(sda=Pin(sda), scl=Pin(scl))
```

G. Código de la clase DHT

```
class DHT:
    def __init__(self, pin: int):
        self.sensor = dht.DHT22(Pin(pin))


    def measure(self) -> obj:
        return self.sensor.measure()


    def temperature(self) -> float:
        sleep(2)
        self.sensor.measure()
        return self.sensor.temperature()


    def humidity(self) -> float:
        sleep(2)
        self.sensor.measure()
        return self.sensor.humidity()
```

H. Código de la clase accel

```
class accel:
    def __init__(self, i2c, addr=0x68):
        self.iic = i2c
        self.addr = addr
        self.iic.start()
        self.iic.writeto(self.addr, bytearray([107, 0]))
        self.iic.stop()

    def get_raw_values(self):
        self.iic.start()
        a = self.iic.readfrom_mem(self.addr, 0x3B, 14)
        self.iic.stop()
        return a

    def get_ints(self):
        b = self.get_raw_values()
        c = []
        for i in b:
            c.append(i)
        return c

    def bytes_toint(self, firstbyte, secondbyte):
        if not firstbyte & 0x80:
            return firstbyte << 8 | secondbyte
```

```
        return - (((firstbyte ^ 255) << 8) | (secondbyte ^ 255) + 1)


    def get_values(self):
        raw_ints = self.get_raw_values()
        vals = {}
        vals["AcX"] = self.bytes_toint(raw_ints[0], raw_ints[1])
        vals["AcY"] = self.bytes_toint(raw_ints[2], raw_ints[3])
        vals["AcZ"] = self.bytes_toint(raw_ints[4], raw_ints[5])
        vals["Tmp"] = self.bytes_toint(raw_ints[6], raw_ints[7]) / 340.00
+ 36.53
        vals["GyX"] = self.bytes_toint(raw_ints[8], raw_ints[9])
        vals["GyY"] = self.bytes_toint(raw_ints[10], raw_ints[11])
        vals["GyZ"] = self.bytes_toint(raw_ints[12], raw_ints[13])
        return vals


    def val_test(self):
        from time import sleep
        while 1:
            print(self.get_values())
            sleep(0.05)
```