Universidad Politécnica

Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Grado en Ingeniería de Software
Grado en Ingeniería de Computadores

Trabajo Fin de Grado

Análisis e Implementación de un Criptosistema con FPGA: Integración de ECDH y AES-128

Autores: Sofía Covadonga Brotton Ruiz, Diego Andrés Badillo Gómez
Tutor: Vicente A. García Alcántara
Cotutor: Sergio Gil Borrás

Madrid, Noviembre 2024

Resumen

Este proyecto pretende abordar la implementación de un criptosistema utilizando algoritmos de criptografía modernos altamente empleados en la industria para garantizar la integridad y seguridad de la información. En particular, se ha desarrollado un sistema que combina el algoritmo de Diffie-Hellman con curvas elípticas (*Elliptic Curve Diffie-Hellman*, ECDH), para el intercambio de claves, y el algoritmo AES-128 en su modo CTR para el cifrado de datos.

El objetivo principal es explorar y comprobar la viabilidad de aplicar estos algoritmos en dispositivos hardware programables (*Programmable Logic Devices,* PLDs), aunque para ello se realizará un estudio previo de los distintos modos que posee AES. El abordamiento de este problema se llevará a cabo a través de una implementación desde cero de los algoritmos en el lenguaje de programación Python, para su posterior migración al lenguaje de descripción de hardware elegido (*Very High Speed Integrated Circuits Hardware Description Language,* VHDL).

A través de una serie de pruebas, se comparará el rendimiento obtenido de la implementación de los algoritmos en tres etapas: la implementación realizada y librerías criptográficas vs hardware dedicado.

Al final del proyecto se demostrará que los resultados obtenidos demuestran que la combinación de ECDH y AES-128 es efectiva para garantizar una comunicación segura, y que la implementación en FPGA proporciona un rendimiento superior comparado con soluciones basadas únicamente en software.

Abstract

The following project aims to implement a cryptosystem using modern cryptographic algorithms that are widely used in the industry to ensure information integrity and security. Specifically, we have developed a system that combines the Diffie-Hellman algorithm with elliptic curves (ECDH) for the key exchange, and the AES-128 algorithm in CTR mode for data encryption.

Our primary objective is to explore and evaluate the feasibility of applying these algorithms to programmable hardware devices. To accomplish this, we will first conduct a preliminary study of the different AES modes available. The approach to solving this problem involves a ground-up implementation of the algorithms in Python, followed by their subsequent migration to the selected hardware description language (VHDL).

Through a series of tests, the performance of the algorithm implementations will be compared across three stages: custom implementations, cryptographic libraries, and dedicated hardware.

By the end of the project, we expect the results to show that the combination of ECDH and AES-128 effectively ensures secure communication, and that the FPGA implementation delivers superior performance compared to software-only solutions.

ÍNDICE

1. Introducción **1**
1.1 Objetivos de la investigación 2
1.2 Estructura de la memoria 3
2. Estado del arte **4**
2.1 Evolución reciente de la criptografía post-AES 4
2.2 Avances en criptografía de curvas elíptica 5
2.3 Implementaciones hardware de algoritmos criptográficos 6
2.4 Desafíos actuales en seguridad criptográfica 7
2.5 Trabajos relacionados 9
3. Metodología **12**
3.1 Enfoque general de la investigación 12
3.2 Herramientas de software empleadas 13
3.2.1 Entorno de desarrollo: Visual Studio Code 14
3.2.2 Lenguaje de programación: Python 14
3.2.3 Plataforma de diseño hardware: Vivado 14
3.2.4 Lenguaje de descripción de hardware: VHDL 14
3.2.5 Consideraciones sobre la implementación 15
3.3 Métodos de evaluación y análisis 15
3.4 Limitaciones del estudio 16
4. Criptografía simétrica y asimétrica **18**
4.1 Limitaciones 18
5. AES y sus modos de operación **21**
5.1 Introducción a AES 21
5.2 Explicación del algoritmo 22
5.3 Modos de operación 27
5.3.1 ECB (Electronic Codebook) 27
5.3.2 CBC (Cypher Block Chaining) 29
5.3.3 CTR (Counter) 30
5.3.4 Otros modos no autenticados (CFB, OFB) 31
5.3.5 Modos autenticados (GCM y CCM) 32
6. Implementación de AES **33**
6.1 Detalles generales de implementación 33
6.2 Explicación del código 35
6.3 Pruebas, validación y análisis comparativo 37
6.3.1 Vectores de prueba y verificación 37
6.3.2 Pruebas de rendimiento 38
6.3.3 Pruebas de seguridad 39
6.3.4 Comparación de resultados 40
7. Curvas elípticas y Diffie-Hellman **50**
7.1. Algoritmo de Diffie-Hellman (DH) 50
7.2 Curvas elípticas en la criptografía 51

7.3 Diffie-Hellman con curvas elípticas (ECDH) 55
7.3.1 Parámetros de las curvas 56
7.3.2 Obtención de claves 58
7.4. Matemáticas de las curvas elípticas 58
7.4.1 Suma de puntos 58
7.4.2 Multiplicación de un punto por un escalar 61
7.4.3 Operaciones de Montgomery 63
7.5 Problemas y desventajas de utilizar ECDH 63
7.6 Implementación y pruebas en Python 65
7.6.1 Detalles de implementación 65
7.6.2 Función de derivación de claves (KDF) 66
7.6.3 Desafíos y errores 66
7.6.4 Pruebas y verificación 67
8. Implementación criptográfica en FPGA 71
8.1. Implementación en el Hardware 71
8.1.1 Descripción del hardware 71
8.1.2 FPGA vs CPU 73
8.2 Diseño y arquitectura 75
8.2.1 Estructura de Alto nivel 75
8.2.2 Componentes del sistema y módulos 75
8.3 Optimización 79
8.3.1 Mejora sobre hardware 79
8.4 Validación y pruebas 82
8.4.1 Test Unitarios. Pruebas por Módulos. 82
8.4.2 Pruebas de rendimiento 89
8.4.3 Análisis de errores 91
9. Análisis de seguridad y ataques potenciales 95
9.1 Ataques contra AES 95
9.2 Ataques contra ECDH 97
9.2.1 Ataques al logaritmo discreto [103] 97
9.3.2. Ataque de claves débiles 103
9.3.3. Ataque en protocolos, Man in the middle (MitM) 104
9.3 Ataques históricos relevantes 105
9.3.1 Ataque de Lentra contra ECDSA 105
9.3.2 Ataque de Dual_EC_DRBG 106
9.4 Ataques de canal lateral 106
9.5 Consideraciones de seguridad para implementaciones en hardware 108
10. Aspectos prácticos y aplicaciones reales 110
11. Impacto social y medioambiental 112
12. Conclusión 114
12.1. Líneas futuras 115
13. Referencias 117
APÉNDICES 127
Apéndice A 128

A.1 Descripción del ejemplo 128
A.2 Expansión de Clave 128
A.3 Rondas del Cifrado 1127
A.4 Resultado Final 130
Apéndice B 131
B.1 Suma de Puntos 131
B.3 Doble de Puntos 131
B.3 Multiplicación Escalar 132
B.4 Reducción Modular 132
B.5 Conversión de Coordenadas 133

Lista de figuras

Figura 1: *Diagrama general de AES* 21
Figura 2: *Matriz State Array* 22
Figura 3: *Tabla S-box* 24
Figura 4: *Diagrama de la operación MixColumns* 24
Figura 5: *Diagrama de cifrado AES* 26
Figura 6: *Comparación de cifrado ECB contra otros modos* 27
Figura 7: *Diagrama de cifrado de bloques CBC* 28
Figura 8: *Diagrama de cifrado de bloques CTR* 29
Figura 9: *Prueba de verificación en modo ECB* 36
Figura 10: *Prueba de verificación en modo CBC* 37
Figura 11: *Prueba de verificación en modo CTR* 37
Figura 12: *Prueba de seguridad en modo ECB* 42
Figura 13: *Pruebas de seguridad en modo CBC* 44
Figura 14: *Prueba de seguridad en modo CTR* 45
Figura 15: *Prueba estadística en modo ECB* 46
Figura 16: *Prueba estadística en modo CBC* 46
Figura 17: *Prueba estadística en modo CTR* 47
Figura 18: *Representación de las curvas elípticas* 52
Figura 19: *Diagrama de intercambio de claves Diffie-Hellman* 54
Figura 20: *pseudocódigo algoritmo suma de dos puntos de la curva* 59
Figura 21: *Suma de dos puntos de la curva elíptica* 59
Figura 22: *Pseudocódigo algoritmo de multiplicación de un punto por un escalar* 60
Figura 23: *Prueba de integración de la curva p256* 65
Figura 24: *Prueba de integración de la curva 25519* 66
Figura 25: *Prueba de seguridad para la curva p256* 66
Figura 26: *Prueba de seguridad para la curva 25510* 66
Figura 27: *Prueba de caso límite sobre la curva p256* 67
Figura 28: *Prueba de caso límite sobre la curva 25519* 67
Figura 29: *Prueba de rendimiento de la curva p256* 67
Figura 30: Prueba de rendimiento de la curva 25519 67
Figura 31: *FPGA AMD-Xilinx Basys 3* 69
Figura 32: *FPGA AMD-Xilinx Zybo Zynq-7000* 70
Figura 33: *FPGA AMD-Xilinx Nexys A7* 71
Figura 34: *Diseño del criptosistema* 73
Figura 35*: Diseño del módulo ECCDH* 74

Figura 36: *Diseño del módulo AES* 75
Figura 37: *Diseño del módulo UART* 76
Figura 38: *Diseño del módulo de control* 77
Figura 39: *Prueba en Python 3 usando xgcd* 80
Figura 40: *Prueba en Vivado usando la implementación xgcd* 80
Figura 41: *Prueba en Python 3 usando xgcd para 256bits* 80
Figura 42: *Prueba en Vivado usando la implementación xgcd para 256bits* 81
Figura 43: *Prueba con P = O, Q= G en Python* 81
Figura 44: *Prueba con P = O, Q= G en Vivado* 81
Figura 45: *Prueba con P y Q = G en Python* 81
Figura 46: *Prueba con P y Q = G en Vivado* 82
Figura 47: *Prueba con Q = G y P = K en Python* 82
Figura 48: *Prueba con Q = G y P = K en Vivado* 82
Figura 49: *Prueba multiplicación punto por escalar en Python* 82
Figura 50: *Prueba multiplicación punto por escalar en Vivado* 82
Figura 51: *Prueba de multiplicación punto por escalar con 256 bits en Python* 83
Figura 52: *Prueba de multiplicación punto por escalar con 256 bits en Vivado* 83
Figura 53: *Prueba de generación de clave privada d, $1 \leqq d \leqq n-1$* 83
Figura 54: *Prueba de validación de puntos en Python* 84
Figura 55*: Prueba de validación de puntos en Vivado* 84
Figura 56: *Prueba de validación de puntos con P fuera de la curva en Python* 84
Figura 57: *Prueba de validación de puntos con P fuera de la curva en Vivado* 84
Figura 58: *Prueba del módulo AES* 85
Figura 59: *Prueba del módulo AES transmitiendo mensajes* 85
Figura 60: *Prueba de funcionamiento de la UART (Fuente DigiKey TechForum)* 86
Figura 61*: Prueba del módulo de control (1)* 86
Figura 62: *Prueba del módulo de control (2)* 87
Figura 63: *Prueba de rendimiento del módulo ECDH en vivado (1)* 87
Figura 64: *Prueba de rendimiento del módulo ECDH en vivado (2)* 87
Figura 65: *Prueba de rendimiento del módulo AES en vivado (1)* 88
Figura 66: *Prueba de rendimiento del módulo AES(2)* 88
Figura 67: *Pseudo algoritmo del ataque Baby-Step Giant-Step* 96

Lista de tablas

Tabla 1: *Prueba de rendimiento de AES en Python* 40

1. Introducción

La criptología, definida como el estudio de técnicas para la comunicación segura, abarca tanto la criptografía como el criptoanálisis. Sus orígenes se remontan a miles de años atrás, concretamente al antiguo Egipto, donde se utilizaban jeroglíficos para ocultar mensajes. Sin embargo, fue en el siglo XX, cuando se produjo una transformación significativa impulsada por la necesidad de mejoras debido a conflictos globales. Una innovación clave fue la invención del RSA en el 1977, que revolucionó la comunicación segura utilizando encriptado y desencriptado con dos claves.

En la era digital actual, la criptología se ha convertido en algo integral en la protección de datos, especialmente en un mundo tan interconectado como el nuestro. En 2001, se adoptó como estándar el AES [1] *(Advanced Encryption Standard*), que ofrecía un encriptado robusto para sustituir al DES (*Data Encryption Standard*). Paralelamente, el desarrollo de la criptología cuántica, con su promesa teórica de encriptado a través de QKD (*Quantum Key Distribution*), ha abierto nuevas fronteras en la seguridad de la información.

Este proyecto se centra en AES y su potencial en el entorno actual, donde las amenazas cibernéticas evolucionan constantemente. AES se mantiene como el algoritmo más importante a día de hoy, ofreciendo un alto nivel de seguridad que se considera resistente incluso ante las futuras capacidades de la computación cuántica. Su excelente rendimiento, tanto en software como en hardware, junto con su desarrollo basado en la colaboración internacional, lo convierten en un objeto de estudio fundamental para la innovación en criptología.

La investigación no solo profundiza en los aspectos teóricos y prácticos de AES, sino que también explora su integración con el protocolo DIffie-Hellman [2] basado en curvas elípticas. Esta combinación es particularmente relevante en la criptografía moderna, ya que fusiona la eficiencia de AES, con la seguridad del intercambio de claves, abordando así una de las principales vulnerabilidades de los sistemas criptográficos simétricos.

El trabajo realizado en software sienta las bases para la posterior implementación en hardware [3], específicamente en matriz de puertas programable (*Field-Programmable Gate Arrays*, FPGAs) [4]. Esta transición es crucial, debido a que permite explorar optimizaciones y ventajas que ofrece el hardware dedicado, potencialmente mejorando tanto su velocidad

como su seguridad en aplicaciones del mundo real. Las FPGAs ofrecen una flexibilidad única que permite optimizar el diseño para máximo rendimiento y eficiencia. Además, ofrece una capa adicional de seguridad al aislar físicamente el proceso de encriptación, lo que la hace más resistente a ciertos tipos de ataques. Por último, esta implementación permite explorar la viabilidad de sistemas criptográficos reconfigurables, que serían capaces de adaptarse rápidamente a nuevas amenazas. Esta parte de la investigación no solo busca mejorar el rendimiento, sino también abrir nuevas posibilidades en el diseño de sistemas de seguridad robustos y adaptables.

En resumen, esta investigación busca contribuir al conocimiento actual sobre AES, explorando su eficacia, eficiencia y adaptabilidad en el panorama tecnológico en constante evolución. Aunque se basa en técnicas y algoritmos establecidos, aporta valor a través de su enfoque integral y práctico, así como la implementación en un hardware dedicado.

1.1 Objetivos de la investigación

Los principales objetivos de este proyecto son:

1. Realizar una implementación desde cero de los algoritmos criptográficos que componen el criptosistema, específicamente AES y Diffie-Hellman con curvas elípticas (ECDH) [5], sin utilizar librerías externas. Esta decisión es crucial ya que la implementación en hardware requiere que el código sea escrito en VHDL puro.
2. Profundizar en la investigación y comprensión de los algoritmos y sus fundamentos matemáticos para asegurar la correcta implementación de los mismos.
3. Desarrollar varios modos de operación de AES y comparar su rendimiento y seguridad para seleccionar el más adecuado para la implementación en FPGA.
4. Comparar la implementación con librerías criptográficas establecidas. Esto permitirá visualizar la diferencia de tiempo entre la implementación y las librerías ya establecidas, así como evaluar la eficiencia y rendimiento de la solución propuesta.
5. Realizar primero una implementación en Python y posteriormente migrar los algoritmos a lenguaje de descripción hardware (VHDL) para su uso en FPGA, aplicando este proceso tanto para AES como para ECDH.
6. Comparar detalladamente el rendimiento del criptosistema en dos plataformas distintas: un ordenador con CPU de uso general y una FPGA con uso específico para criptografía. Esta comparación ayudará a determinar las ventajas e inconvenientes de utilizar un hardware dedicado para operaciones criptográficas, considerando factores como el tiempo de ejecución y la eficiencia energética.

7. Optimizar los algoritmos implementados en FPGA para mejorar su rendimiento, explorando técnicas como la paralelización de operaciones y la optimización del uso de recursos.
8. Explorar la viabilidad de sistemas criptográficos reconfigurables, que serían capaces de adaptarse rápidamente a nuevas amenazas, abriendo nuevas posibilidades en el diseño de sistemas de seguridad robustos y adaptables.

1.2 Estructura de la memoria

Para abordar estos temas, este documento se estructura de la siguiente manera: Tras la introducción, se presenta una revisión detallada del algoritmo AES, sus principales características y alguno de sus modos de operación principales. Después, se realiza un análisis de la implementación en Python con los modos escogidos, seguido de una exploración de la criptografía de curvas elípticas y el protocolo Diffie hellman, examinando su integración con AES. Posteriormente, se realiza una comparativa con implementaciones existentes y se analizan posibles ataques. El estudio termina con la implementación AES con el modo escogido y la curva seleccionada explorando las optimizaciones y ventajas del hardware dedicado. Finalmente, se realiza un estudio de las aplicaciones prácticas, las posibles mejoras y líneas futuras, así como consideraciones éticas, sociales y medioambientales.

2. Estado del arte

Desde la adopción del AES, el campo de la criptografía ha continuado evolucionando, impulsado por nuevas tecnologías, la aparición de nuevos ataques y la creciente necesidad de algoritmos más seguros y eficientes. Esta sección examina los avances más relevantes en el ámbito de la criptografía moderna, centrándose específicamente en aquellos aspectos que afectan al trabajo.

2.1 Evolución reciente de la criptografía post-AES

La computación cuántica, surgió como concepto en la década de 1980 pero ha visto avances significativos en los últimos años, transformando radicalmente el panorama criptográfico. Con el desarrollo de algoritmos cuánticos como el algoritmo de Shor, muchos sistemas criptográficos tradicionales, incluyendo RSA y ECC (*Elliptic Curve Cryptography*) se han vuelto potencialmente vulnerables. En respuesta a estos avances, la NIST (*National Institute of Standards and Technology*, Instituto Nacional de Estándares y Tecnología de Estados Unidos) ha tomado la iniciativa de estandarizar algoritmos criptográficos resistentes a ataques cuánticos [6]. Algunos de los candidatos más prometedores incluyen criptosistemas basados en retículos, en *hash* [7], y de polinomios multivariados [8]. Aunque AES se considera seguro frente a estos ataques cuánticos, se están buscando formas de integrarlo con algoritmos post-cuánticos [9] para mejorar su resistencia a largo plazo.

Paralelamente, la encriptación homomórfica ha emergido como una tecnología prometedora [10]. Esta técnica permite realizar cálculos sobre datos cifrados sin necesidad de descifrarlos primero, ofreciendo seguridad para el procesamiento de datos en entornos no seguros. Se están explorando formas de integrar AES con esquemas homomórficos, mejorando así la seguridad en entornos de la nube, manteniendo la eficiencia y robustez que caracteriza AES.

En el ámbito de la tecnología *blockchain* [11], se han desarrollado las pruebas de conocimiento cero *(Zero-knowledge proofs*), las cuales permiten verificar la posesión de información sin revelar el contenido de la misma. Esta innovación ha supuesto una mejora significativa en la privacidad y seguridad digital en aplicaciones *blockchain*, abriendo nuevas posibilidades para la integración de AES en sistemas descentralizados.
Por otro lado, los modos de operación de AES también han evolucionado para adaptarse a nuevos requisitos de seguridad y rendimiento. Se ha observado una creciente adopción de

modos autenticados como GCM (*Galois/Counter Mode*) y CCM (*Counter with CBC-MAC*) que garantizan tanto confidencialidad como integridad. Para aplicaciones que requieren de un alto rendimiento, se utilizan modos como CTR (*Counte*r) o XTS (*XEX-based tweaked-codebook mode with ciphertext stealing*). Además, se han desarrollado modos más seguros como OCB (*Offset Codebook Mode*) y SIV (*Synthetic Initialization Vector*).

Finalmente, con el avance en el ámbito del Internet de las Cosas (IoT), AES se ha adaptado para ofrecer soluciones ligeras y eficientes. Los modos GCM y CCM, en particular, han demostrado ser adecuados para dispositivos con recursos limitados, permitiendo que AES siga siendo relevante en este nuevo entorno.

2.2 Avances en criptografía de curvas elíptica

La ECC ha surgido como una alternativa poderosa frente a RSA. ECC ofrece seguridad comparable con RSA, pero con claves significativamente más pequeñas, haciéndola más eficiente y particularmente útil para dispositivos con poder computacional limitado. Esta eficiencia ha llevado a su adopción en diversos entornos desde comunicaciones seguras hasta tecnología blockchain.

En los últimos años se han desarrollado nuevos tipos de curvas elípticas que mejoran las ya utilizadas. Las curvas Edwards [12], por ejemplo, ofrecen una solución más rápida y segura comparada con curvas como secp256r. Similarmente, las curvas de Montgomery [13], como Curve25519, han sido optimizadas para operaciones de alta velocidad. Estos avances han sido cruciales para la implementación de ECC en aplicaciones del mundo real.

La comunidad criptográfica ha sido proactiva en asegurar la seguridad de implementaciones ECC. El criptógrafo Daniel J. Bernstein [14] inició un proyecto llamado *SafeCurves* para evaluar y promover curvas seguras, que ha sido esencial para crear conciencia sobre posibles vulnerabilidades en algunas claves. Por otro lado, el NIST, trabaja continuamente en la actualización y estandarización de ECC para cumplir con los requisitos de seguridad actuales.

La amenaza potencial de computación cuántica ha generado una oleada de investigación en variaciones ECC resistentes a estos ataques. Una interesante solución implica el uso de curvas isogénicas, como se ven en el protocolo SIKE (*Supersingular Isogeny Key Encapsulation*). Aunque SIKE ha sido recientemente comprometido, esta área de

investigación sigue en desarrollo, subrayando los desafíos continuos en la criptografía post-cuántica [15].

En el área de mensajería segura, el algoritmo *Double Ratchet* [16], que suele incorporar ECC, ha revolucionado la seguridad de protocolos de comunicación. Este algoritmo asegura que el compromiso de claves actuales no afecte la seguridad de mensajes futuros y pasados, que es algo crítico en aplicaciones de mensajería modernas como WhatsApp [17].

ECC también está siendo ampliamente utilizado en protocolos de intercambio de claves [18], destacando el ECDH (*Elliptic Curve DIffie-Hellman*), que ha llamado la atención para este estudio y para complementar el algoritmo AES. Esta integración combina la eficiencia del cifrado simétrico (que se analizará en detalle en una sección posterior) con la seguridad de los sistemas de intercambio de claves de clave pública.

2.3 Implementaciones hardware de algoritmos criptográficos

Las implementaciones hardware de algoritmos criptográficos ofrecen ventajas significativas en términos de rendimiento y seguridad. Aunque existen varias tecnologías hardware relevantes para la criptografía, el proyecto se centra principalmente en las FPGAs, aunque se mencionan brevemente otras tecnologías relacionadas.

Las FPGAs son componentes de hardware versátiles que pueden ser programados para realizar tareas específicas, haciéndolas altamente útiles para aplicaciones criptográficas. Específicamente, pueden explotar el paralelismo y el *pipelining* (técnica de procesamiento en cadena que permite ejecutar múltiples operaciones simultáneamente) de algunos algoritmos para reducir el tiempo de procesamiento. Además, las FPGAs pueden actuar como aceleradores de hardware al implementar directamente en su estructura circuitos optimizados para algoritmos específicos, lo cual obtiene un rendimiento significativamente superior al de las implementaciones en software.

Actualmente, las FPGAs se están utilizando para implementar y evaluar algoritmos post-cuánticos, como los basados en retículos o en isogénicas, mencionados en los puntos anteriores. Esto ha proporcionado información valiosa sobre su viabilidad y rendimiento. La flexibilidad de las FPGAs para ser reprogramadas las hace ideales en este entorno criptográfico en constante evolución.

Las FPGAs además ofrecen resistencia a ataques de canal lateral [19], una vulnerabilidad que afecta a muchos criptosistemas. Esta resistencia se debe a su capacidad para implementar contramedidas contra el análisis diferencial de consumo (*Differential Power Analysis*, DPA) y el análisis electromagnético (*Electromagnetic Analysis*, EMA).

Más allá de las FPGAs, existen otras tecnologías hardware relevantes para la criptografía, aunque muchas de ellas son más especializadas y menos versátiles que las FPGAs por su papel significativo en implementaciones criptográficas modernas. Los Procesadores de Seguridad Dedicados, como el Apple T2 [20], ofrecen soluciones de seguridad integradas para dispositivos específicos. Por otro lado los Enclaves Seguros, ejemplificados por Intel SGX [21] y ARM TrustZone [22], proporcionan entornos de ejecución aislados para operaciones criptográficas sensibles dentro de procesadores de propósito general. Aunque aún en etapas tempranas, los Aceleradores Cuánticos de Propósito Especial prometen avances significativos en criptografía post-cuántica. Por último, los TPMs (*Trusted Platform Modules*) [23] siguen siendo fundamentales para el arranque seguro y la protección de credenciales en dispositivos modernos.

En conclusión, las implementaciones en hardware representan el futuro de la criptografía no sólo por la velocidad mencionada y su flexibilidad, si no también por su menor latencia y mayor *throughput* (rendimiento) en comparación con el software ejecutado en CPUs de propósito general.

2.4 Desafíos actuales en seguridad criptográfica

Como ya se ha mencionado varias veces, la computación cuántica representa un desafío significativo para la criptografía actual. Se necesita estandarizar algoritmos y medidas para contrarrestar estos potenciales ataques, debido al poder devastador que podrían llegar a tener [24]. Esto también se relaciona con la necesidad de que la criptología se vuelva más ágil y logre adaptarse para compensar la velocidad con la que surgen nuevos ataques y amenazas. La magnitud de este desafío se refleja en las estimaciones de *Boston Consulting Group*, que proyecta que para 2030, el mercado de la computación cuántica alcanzará los $65 mil millones, subrayando la urgencia de desarrollar e implementar criptografía post-cuántica [25].

Aunque se ha mejorado la resistencia a los ataques de canal lateral (*side-channel attacks*), estos siguen siendo una preocupación sin resolver totalmente. Estos ataques explotan

características físicas de los sistemas criptográficos, como el consumo de energía o las emisiones electromagnéticas, para extraer información sensible. Es necesario aumentar la seguridad, lo cual podría lograrse mediante implementaciones hardware más robustas. Además, es crucial desarrollar técnicas para detectar y mitigar estos ataques en tiempo real. Aunque las cifras exactas son difíciles de obtener, algunos estudios sugieren que estos ataques representan un porcentaje significativo de las amenazas a sistemas criptográficos.

Con el aumento de la tecnología *blockchain*, han surgido nuevos desafíos de seguridad. Un problema significativo es el llamado "ataque del 51%", donde una entidad que controla más de la mitad del poder computacional de la red puede potencialmente manipular la *blockchain* [26]. Actualmente se está investigando cómo crear mecanismos de consenso más robustos, así como mejorar la escalabilidad y seguridad general de los sistemas *blockchain*. La gravedad de este problema se evidencia en los datos de *Chainalysis*, que reporta que en 2022 se registraron más de 200 ataques a plataformas *blockchain*, resultando en pérdidas de más de $3 mil millones [27].

El creciente uso de la Inteligencia Artificial (IA) en criptografía presenta tanto oportunidades como desafíos. Por un lado, las técnicas de IA pueden usarse para detectar vulnerabilidades y mejorar los sistemas criptográficos. Sin embargo, también podrían ser utilizadas por atacantes para explotar debilidades en estos sistemas. Este es un punto crucial a considerar en el desarrollo de futuras implementaciones criptográficas, buscando aprovechar las ventajas de la IA mientras se mitigan sus potenciales riesgos.

Estos desafíos, aunque algunos no afectan directamente a AES, son cruciales para entender el panorama actual de la criptografía. AES, siendo un pilar fundamental en la seguridad de datos, debe considerarse en el contexto de estas amenazas emergentes. Por ejemplo, aunque AES se cree que es resistente a ataques cuánticos, su integración con otros sistemas y protocolos debe tener en cuenta estas nuevas amenazas.

2.5 Trabajos relacionados

El campo de la implementación de algoritmos criptográficos en FPGAs es extenso y se investiga activamente. A través de una revisión exhaustiva de la literatura, se han identificado numerosos estudios relevantes que abordan diversos aspectos de este campo.

Efficient FPGA Implementation of Blockchain Operation, de Eke Timur, Schapeler Nicolas, Starnecker Christoph y Weiser Florian [28], aborda la implementación de RSA y SHA-256 en FPGA para su uso en Blockchain. Este estudio logró implementar un algoritmo de minado de Bitcoin en FPGA, alcanzando resultados competitivos en comparación con CPUs y mayor eficiencia energética que las GPUs.

A New Crypto-based Processor System Design Using FPGA, publicado por Suravi Lahiri, Kaustav M. Chatterjee y Swarup K. Mitra [29], se centra en el diseño de un procesador criptográfico utilizando RSA, demostrando la versatilidad de las FPGAs en aplicaciones criptográficas específicas.

En el ámbito de AES, *Encryption and decryption using FPGA* de Nikhilesh Nayak, Akshay Chandak, Nisarg Shah y B Karthikeyan [30] compara las implementaciones de RSA y AES en FPGA. Por otro lado, *FPGA Implementation of AES Encryption and Decryption* de Sounak Samanta [31] se enfoca exclusivamente en AES, utilizando una arquitectura de pipeline para optimizar el rendimiento.

Profundizando en las implementaciones de AES, el estudio de T. Pöppelmann y T. Güneysu, *High-Speed and Area-Efficient FPGA Implementations of AES-GCM for 100G Ethernet* (2019) [32], aborda la implementación de AES en modo GCM para aplicaciones de alta velocidad. Este trabajo es particularmente relevante para comprender las optimizaciones posibles en implementaciones de AES para entornos de alto rendimiento.

Además, A. Moradi et al. en *Lightweight and Fault-resistant FPGA Implementation of the Advanced Encryption Standard* (2011) [33] abordan no sólo la eficiencia sino también la resistencia a fallos en la implementación de AES, un aspecto crucial para la seguridad en aplicaciones del mundo real. Complementando este enfoque, T. Güneysu y A. Moradi en *'Side-Channel Attack Resistant FPGA Implementation of AES*' (2011) [34] se centran específicamente en la resistencia a ataques de canal lateral, un aspecto crítico de la seguridad que se ha mencionado anteriormente en la sección estado del arte.

Un estudio particularmente relevante para el proyecto es *FPGA implementation of a wireless sensor node with built-in security coprocessors for secured key exchange and data transfer* de Abdelmoughni Toubal, Billel Bengherbia, Mohamed Ould Zmirli y Abderrezak Guessoum [35]. Este trabajo implementa un criptosistema basado en AES-128, curvas elípticas (ECC-163) y SHA-256, proporcionando una base sólida para la investigación.

FPGA based Blockchain System for Industrial IoT [36] también utiliza un criptosistema similar, combinando SHA256, ECDSA, ECIES y KDF, e integra un entorno de ejecución confiable (TEE) basado en blockchain en una FPGA.

En cuanto a la criptografía post-cuántica, *FPGA Implementation of Post-Quantum DME Cryptosystem* de Jose L. Imaña e Ignacio Luengo [37] presentan una arquitectura de alto rendimiento para DME-(3, 2, 48), abordando los desafíos emergentes de la computación cuántica.

Adicionalmente, es importante mencionar el trabajo de Khalil Zeraoulia y Mounir Lallali, *FPGA-based implementation of elliptic curve Diffie-Hellman key exchange protocol* [38], que se centra específicamente en la implementación de ECDH en FPGA. Este estudio es particularmente relevante para el proyecto, ya que combina el intercambio de claves Diffie-Hellman con curvas elípticas en una implementación hardware.

En el ámbito de la criptografía de curvas elípticas, K. Javeed y X. Wang presentan en *Efficient FPGA Implementation of Elliptic Curve Cryptography over Prime Fields* (2016) [39] una implementación eficiente de operaciones de curva elíptica en FPGAs. Este trabajo es directamente relevante para la implementación de ECDH, proporcionando perspectivas valiosas sobre optimizaciones posibles.

Otro estudio significativo es *Efficient FPGA Implementation of Elliptic Curve Cryptography and SHA-3 Algorithms* de Santosh Ghosh, Dipanwita Roychowdhury y Abhijit Das [40], que presenta una implementación eficiente de criptografía de curva elíptica junto con el algoritmo de *hash* SHA-3 en FPGA, proporcionando insights valiosos sobre la optimización de estos algoritmos en hardware.

Para concluir, la investigación en la implementación de algoritmos criptográficos en FPGAs es extensa y variada. Estos estudios han demostrado las ventajas de usar FPGAs para mejorar el rendimiento, la eficiencia energética y la seguridad de los algoritmos criptográficos. Desde la combinación de RSA y SHA-256, hasta la implementación de AES y ECDH para blockchain, estos proyectos proporcionan una base sólida y motivación para la implementación y optimización de criptosistemas en FPGAs [41].

La capacidad de las FPGAs para manejar operaciones paralelas y su eficiencia en el uso de recursos las hace una elección ideal para aplicaciones criptográficas. El proyecto, que combina AES con Diffie-Hellman y curvas elípticas, se basa en estos trabajos previos y busca contribuir al campo con una implementación integrada y eficiente de estos algoritmos en FPGA.

3. Metodología

Esta sección describe el enfoque metodológico utilizado en la investigación. Las subsecciones siguientes elaboran los aspectos específicos del diseño del proyecto, detallando los procesos y herramientas empleados, y asegurando la transparencia y reproducibilidad de los resultados.

3.1 Enfoque general de la investigación

Para este trabajo, debido a su naturaleza, se ha utilizado una metodología principalmente experimental y cuantitativa [42]. El enfoque experimental se ha empleado para implementar AES en sus diferentes modos de operación, ECDH, realizar tests e implementar el algoritmo escogido en la FPGA. Este aspecto constituye el núcleo cuantitativo del estudio donde se han utilizado y comparado métricas específicas de rendimiento y seguridad.

La investigación aplicada se refleja en la revisión de literatura, el análisis de los resultados y su interpretación. Esta parte del proyecto implicó un estudio profundo de los fundamentos teóricos y las aplicaciones prácticas de los algoritmos implementados.

Pasos generales que se han seguido y como se relacionan con los objetivos del trabajo:

1. Revisión de literatura: Investigación sobre AES, ECDH e implementaciones en FPGA, incluyendo estudios recientes y mejores prácticas en el campo.
2. Fase de implementación de software: Desarrollo de AES en Python implementando varios modos de operación, implementación de ECDH en Python y finalmente, diseño y ejecución de tests de funcionalidad y rendimiento.
3. Análisis de resultados de software: Comparación de rendimiento entre los diferentes modos de AES, comparación contra implementaciones estándar de librerías criptográficas y por último, evaluación de la seguridad y eficiencia de la implementación de ECDH.
4. Selección para implementación hardware: Incluye la evaluación de los resultados y la comparativa realizada anteriormente, elección del modo de AES más adecuado para FPGA y consideración de limitaciones y ventajas del hardware objetivo.
5. Implementación en hardware: Traducción del algoritmo seleccionado y las dos implementaciones de ECDH a VHDL, diseño de la arquitectura hardware y programación de la FPGA.

6. Tests y optimización en hardware: Primero se comprueba cada parte del criptosistema, evaluando el correcto funcionamiento por separado de los algoritmos de ECDH y AES. Posteriormente, se evalúa el conjunto del sistema habiendo conectado ambos algoritmos, siendo la última etapa la comprobación del rendimiento frente a sus homólogos ejecutándose en una CPU y comprobando sus tiempos de ejecución frente a las soluciones existentes.
7. Análisis comparativo final.
8. Discusión de resultados, conclusiones y posibles direcciones para futuras investigaciones.

Es importante destacar el porqué de la implementación desde cero de los algoritmos. Por un lado, se quería comprender completamente cómo funcionaban internamente y demostrar este entendimiento. Al hacer una implementación de cero, se alcanza este objetivo. Si bien se usaron librerías de criptografía para algunos de los tests de seguridad, la implementación principal se realizó sin dependencias externas a librerías de criptografía. Además, esta implementación desde cero facilitó el trabajo posterior de traducción a VHDL para la implementación en hardware.

Esta metodología se apoya sobre un marco teórico basado en principios de la criptografía moderna [43]. Este marco proporciona la base para entender la importancia y el funcionamiento de los algoritmos implementados. En concreto, los conceptos claves que fundamentan la investigación son: Los fundamentos matemáticos de AES y la criptografía de curvas elípticas, la teoría detrás de los diferentes modos de operación de AES, los principios de la criptografía de clave pública y los protocolos de intercambio de claves y finalmente, los conceptos básicos de la implementación de algoritmos en hardware. Estos fundamentos han guiado las decisiones desde la elección de los algoritmos hasta el diseño de los experimentos y análisis.

3.2 Herramientas de software empleadas

En este trabajo se utilizaron diversas herramientas software. Todo el código desarrollado para este proyecto se puede encontrar en el siguiente repositorio de GitHub: BadilloBrotton/ECCDH-AES/

A continuación, se detallan las principales empleadas:

3.2.1 Entorno de desarrollo: Visual Studio Code

Para la implementación de los algoritmos en Python (versión 3.x), tanto de AES con sus respectivos modos de operación como para Diffie-Hellman con curvas elípticas, se empleó el entorno de desarrollo (IDE) Visual Studio Code [44]. Esta herramienta fue seleccionada por la facilidad de programación y depuración de la que dispone, el control de versiones con herramientas como Git, interfaz de usuario intuitiva y personalizable y finalmente, su amplia gama de extensiones disponibles para Python.

3.2.2 Lenguaje de programación: Python

Python [45] fue elegido como lenguaje de programación principal para el desarrollo del software principalmente por su simplicidad de sintaxis, alta legibilidad del código, amplia gama de librerías criptográficas disponibles (que facilitaron la verificación de la implementación), abundante documentación y su facilidad para realizar pruebas rápidas. Además, se han seguido las guías de estilo PEP8 [46] y utilizado la convención *snake_case* para mantener la coherencia y claridad en el código.

3.2.3 Plataforma de diseño hardware: Vivado

Para la implementación del hardware, se utilizó la plataforma de diseño Vivado [47], desarrollada por Xilinx. Esta herramienta fue escogida por: sus capacidades avanzadas para diseñar, simular e implementar los algoritmos criptográficos en la FPGA, su compatibilidad con el hardware FPGA utilizado en proyecto y por último por sus herramientas de optimización integradas para mejorar el rendimiento del diseño.

3.2.4 Lenguaje de descripción de hardware: VHDL

Para la descripción del hardware y la implementación de los algoritmos en la FPGA, se utilizó VHDL. Es un lenguaje estándar en la industria para este propósito y es además, compatible con la herramienta Vivado y la FPGA utilizada en el proyecto. Adicionalmente, ofrece un alto nivel de abstracción (permitiendo describir el comportamiento del sistema), proporciona estructuras que facilitan la implementación de algoritmos complejos y permite una descripción precisa de la temporalización y el paralelismo utilizada para la optimización de algoritmos.

3.2.5 Consideraciones sobre la implementación

Al inicio del trabajo, se planteó una idea que involucra el uso de un procesador RISC (*Reduced Instruction Set Computing*) [48], concretamente en la arquitectura RISC V, junto con los algoritmos criptográficos. Esta propuesta consistía en realizar una

implementación en C [49] y posteriormente cargarla en una FPGA que tendría en memoria el código de un procesador RISC de código libre, por ejemplo Neorv32 [50]. Este enfoque habría permitido comparar los resultados de un ordenador con arquitectura CISC frente a una arquitectura RISC. El plan incluía el uso de herramientas de traducción de C a VHDL, como *Catapult C Synthesis* [51], para obtener el código necesario para la FPGA. Este código traducido requeriría de una optimización intensiva posterior, ya que estas herramientas de traducción suelen añadir líneas innecesarias que saturan el código y resultan poco eficientes.

Sin embargo, esta idea fue finalmente rechazada por varias razones. Principalmente porque el código VHDL obtenido de las librerías contenía un gran número de líneas de relleno, lo que habría complicado significativamente el proceso de optimización.

En su lugar, se optó por el uso de Python como lenguaje de programación para desarrollar el software de los algoritmos. Sus características no solo facilitaron el desarrollo inicial, sino que también simplificaron la comprobación de la implementación al permitir comparaciones directas con librerías establecidas.

3.3 Métodos de evaluación y análisis

Se han empleado diversos métodos para evaluar y analizar tanto las implementaciones software como hardware de los algoritmos criptográficos.

En cuanto a la implementación software, se ha centrado en métricas clave como el tiempo de ejecución, latencia (tiempo necesario para procesar un bloque de datos), *throughput* (cantidad de datos procesados por unidad de tiempo) [52], uso de memoria y complejidad computacional. Estas métricas han permitido evaluar la eficiencia de los algoritmos en Python, proporcionando una visión completa de su rendimiento en diferentes escenarios.

Por otro lado, para la implementación hardware en FPGA, el enfoque se centró en métricas específicas de este entorno. Se evaluaron las mismas métricas que en el software, pero adaptadas al contexto hardware.

Para la comparación entre software y hardware, se simuló el uso del ordenador para la implementación software y se realizó una simulación del hardware en la FPGA. Debido a que no se podían comunicar directamente en la realidad, se diseñaron pruebas idénticas para

ambas plataformas, permitiendo así una comparación justa de las métricas mencionadas. Esta metodología permitió evaluar el rendimiento relativo de cada implementación en condiciones similares.

La validación y verificabilidad de los resultados fue una prioridad en todo momento. Se utilizaron vectores de prueba estándar reconocidos internacionalmente, lo que permitió verificar la correcta implementación de los algoritmos. Además, se compararon los resultados con implementaciones de referencia de librerías criptográficas establecidas como *PyCryptodome* para Python [53]. Se realizaron numerosas ejecuciones para asegurar la consistencia de los resultados, documentando meticulosamente cada paso para garantizar la reproducibilidad de los experimentos.

Para las pruebas de seguridad sobre el software, se utilizaron herramientas de una librería especializada para implementar el *Statistical Test Suite* (STS) del NIST [54], que permite evaluar la calidad de los generadores de números aleatorios utilizados en criptografía. Además, se analizaron los posibles ataques que podían surgir en este tipo de implementaciones, como ataques de canal lateral y de tiempo, para evaluar la robustez de los algoritmos frente a estas amenazas.

3.4 Limitaciones del estudio

Es importante reconocer las limitaciones del estudio. En cuanto a la velocidad, muchas librerías de software han sido optimizadas durante años, por lo que esta implementación está naturalmente más limitada en este sentido. Aunque se ha intentado optimizar el código todo lo posible, no se pudo implementar técnicas avanzadas de optimización como el uso de instrucciones SIMD (*Single Instruction Multiple Data,* Instrucción Única, Múltiples Datos) [55], en español “una instrucción, múltiples datos”, o la optimización a nivel de ensamblador, que son comunes en librerías profesionales

En cuanto a la seguridad, aunque se han realizado pruebas, los medios disponibles resultan limitados para comprobar exhaustivamente la seguridad de todo el sistema. En un escenario real, una implementación debería hacerse con librerías ya optimizadas y probadas en términos de seguridad, en lugar de una implementación pura. No fue posible realizar pruebas exhaustivas de ataques de canal lateral o análisis diferencial de potencia, que son importantes para evaluar la seguridad de implementaciones criptográficas en hardware.

Por otro lado, los tests se han realizado en un entorno controlado. En el mundo real, pueden surgir problemas adicionales como latencia de red, cargas de datos variables, etc., que no se pudo simular completamente en el estudio. No fue posible simular escenarios de red con alta latencia o pérdida de paquetes, ni situaciones de carga extrema que podrían afectar el rendimiento en un entorno real.

Las limitaciones de la FPGA utilizada (Nexys A7) [56] también son un factor a considerar. Estas incluyen restricciones de espacio, número de celdas lógicas, y bloques DSP disponibles. Además, la velocidad de la frecuencia de reloj está limitada, lo que a su vez condiciona la velocidad general del sistema.

El tiempo y los recursos disponibles han sido condicionantes significativos en este estudio. Aunque se ha centrado en los algoritmos AES y ECDH, el campo de la criptografía es extenso, con muchas otras opciones que no se lograron abordar. Habría sido deseable explorar algoritmos más complejos, incluyendo alternativas post-cuánticas o realizar un análisis comparativo más amplio con modos AES autenticados. Estas limitaciones, sin embargo, no invalidan los resultados obtenidos; en su lugar, proporcionan un contexto adecuado para su interpretación y abren nuevas direcciones para futuras investigaciones y mejoras.

4. Criptografía simétrica y asimétrica

Se ha mencionado el concepto de clave simétrica previamente, y en esta sección se verá que es y en qué se diferencia de la criptografía asimétrica. La criptografía simétrica, también conocida como criptografía de clave secreta, es un método que utiliza una única clave para cifrar y descifrar mensajes entre emisor y receptor. Este enfoque requiere que ambas partes acuerden previamente la clave a utilizar.

Por otro lado, la criptografía asimétrica o de clave pública emplea dos claves relacionadas matemáticamente: una pública para cifrar y una privada para descifrar. Aunque parezcan aleatorias, estas claves están intrínsecamente vinculadas y cumplen funciones complementarias.

Ambos métodos son seguros, pero basan su seguridad en diferentes principios. La criptografía simétrica centra su seguridad principalmente en la clave. La confidencialidad de la clave es crucial, y su longitud es fundamental para resistir ataques por fuerza bruta. Por ejemplo, AES-128 se considera seguro actualmente, mientras que AES-256 ofrece una seguridad aún mayor debido a su espacio de claves significativamente más grande.

La criptografía asimétrica, por su parte, basa su seguridad en problemas matemáticos complejos, como el logaritmo discreto o la factorización de números grandes. Por ejemplo, RSA utiliza la dificultad de factorizar el producto de dos números primos grandes, mientras que los sistemas basados en curvas elípticas (ECC) se apoyan en la complejidad del problema del logaritmo discreto en grupos de curvas elípticas (este tema se explorará en mayor detalle más adelante).

4.1 Limitaciones

Aunque, como ya se ha mencionado, ambos tipos de criptografía son seguros cuando se implementan correctamente, cada uno presenta las limitaciones que, a continuación, se citan:

Limitaciones de la criptografía simétrica:

- Distribución de claves: El principal desafío es el intercambio seguro de la clave inicial. Sin un canal seguro, la clave podría verse comprometida durante la transmisión.

- Escalabilidad: En redes grandes, el número de claves necesarias crece cuadráticamente con el número de participantes $\frac{n(n-1)}{2}$ claves para n participantes, lo que puede volverse difícil de manejar.
- No repudio: Al compartir la misma clave, es difícil probar quién originó un mensaje, lo que puede ser problemático en situaciones que requieren responsabilidad legal.
- Gestión de claves: Mantener y actualizar regularmente un gran número de claves puede ser complejo.

Limitaciones de la criptografía asimétrica:

- Rendimiento: Para una misma longitud de clave y mensaje, los métodos asimétricos requieren significativamente más tiempo de ejecución que los otros.
- Tamaño de las claves: Las claves en criptografía asimétrica deben ser más largas para ofrecer un nivel de seguridad equivalente al de las claves simétricas, lo que aumenta los requisitos de almacenamiento.
- *Overhead* (sobrecarga) de cifrado: El mensaje cifrado suele ocupar más espacio que el original, lo que puede ser un problema en aplicaciones con restricciones de ancho de banda.
- Complejidad: Los algoritmos asimétricos son más complejos de implementar correctamente, lo que puede llevar a vulnerabilidades si no se manejan con cuidado.
- Vulnerabilidad a ataques de intermediario: Los sistemas asimétricos pueden ser vulnerables a ataques de "hombre en el medio" (*Man In The Middle*, MITM).
- Sensibilidad a avances en computación cuántica: Algunos algoritmos asimétricos populares, como RSA, son vulnerables a ataques basados en computación cuántica, lo que ha impulsado la investigación en criptografía post-cuántica.

Estas limitaciones subrayan la importancia de elegir cuidadosamente el tipo de criptografía según las necesidades específicas de cada aplicación, y a menudo conducen a la implementación de sistemas híbridos que aprovechan las fortalezas de ambos enfoques.

En la práctica, muchos sistemas modernos combinan ambos tipos de criptografía. Por ejemplo, en las conexiones HTTPS, se usa criptografía asimétrica para el intercambio inicial de claves y luego se cambia a criptografía simétrica para la comunicación continua, aprovechando así la seguridad de la asimétrica y la eficiencia de la simétrica.

En el contexto del estudio, se ha centrado en AES como ejemplo principal de criptografía simétrica. Como todos los algoritmos simétricos, AES enfrenta el desafío de la distribución segura de claves. Para abordar esta limitación, se integrará AES con el protocolo Diffie-

Hellman, un método de intercambio de claves basado en principios de criptografía asimétrica. Esta combinación permitirá aprovechar la eficiencia de AES mientras se soluciona su principal debilidad.

5. AES y sus modos de operación

En esta sección del proyecto se analizará el Estándar de Encriptación Avanzada (AES) y sus diversos modos de operación. El objetivo para esta sección es proporcionar una comprensión completa de AES, explorando su estructura básica, sus principios de funcionamiento y los modos de operación más utilizados a día de hoy.

5.1 Introducción a AES

AES [57] es un algoritmo de cifrado de clave simétrica por bloques de tamaño 128 bits. Surge de la necesidad de un sistema más seguro que DES (*Data Encryption Standard,* Estándar de Cifrado de Datos) [58]. DES fue creado en 1975 por IBM y se convirtió rápidamente en el método estándar de cifrado. Sin embargo, su tamaño de bloque de 64 bits y sus claves de 56 bits no eran suficientemente robustas ante la evolución de la tecnología. Aunque se desarrollaron soluciones temporales como el Triple DES, que ofrecía mayor seguridad, se necesitaba una solución más eficiente y moderna a largo plazo.

El NIST (*National Institute of Standards and Technology*), en 1997, anunció un concurso para desarrollar un nuevo estándar de cifrado ante los problemas que estaban apareciendo. En 1998, dos criptógrafos belgas, Vincent Rijmen y Joan Daemen, presentaron su algoritmo Rijndael como parte de este concurso [59]. Tras un riguroso proceso de evaluación que duró aproximadamente tres años, en 2001 el algoritmo Rijndael fue seleccionado como el ganador.

Algunas de las características principales que AES debía cumplir eran: ser un cifrado en bloque [60], poder ser utilizado tanto por el gobierno como por la industria, tener una longitud de clave variable [61], y ser más eficiente y resistente que Triple DES. El proceso de selección de AES fue notable por su transparencia y participación internacional, estableciendo un estándar en el desarrollo de estándares criptográficos. AES no solo mejoró la seguridad, sino que también revolucionó la forma en que pensamos sobre la criptografía en la era digital.

A día de hoy, AES es el estándar utilizado en criptografía simétrica y está aprobado por el gobierno estadounidense para proteger información altamente clasificada. Se utiliza ampliamente en diversas aplicaciones, desde la seguridad en comunicaciones por Internet hasta el cifrado de archivos, y ha influido significativamente en el desarrollo de otros algoritmos y protocolos criptográficos

5.2 Explicación del algoritmo

En esta sección, se examinarán las características fundamentales de AES, su estructura básica y los pasos que conforman su proceso de cifrado. Comprender a fondo el funcionamiento interno de AES es el primer objetivo crucial para la implementación. Este entendimiento detallado permite: desarrollar implementaciones más eficientes optimizando cada paso del algoritmo, identificar posibles vulnerabilidades rápidamente y tomar medidas para corregirlas, tomar decisiones informadas para conseguir un equilibrio entre rendimiento y seguridad. En el siguiente diagrama se puede observar la estructura general del algoritmo AES:

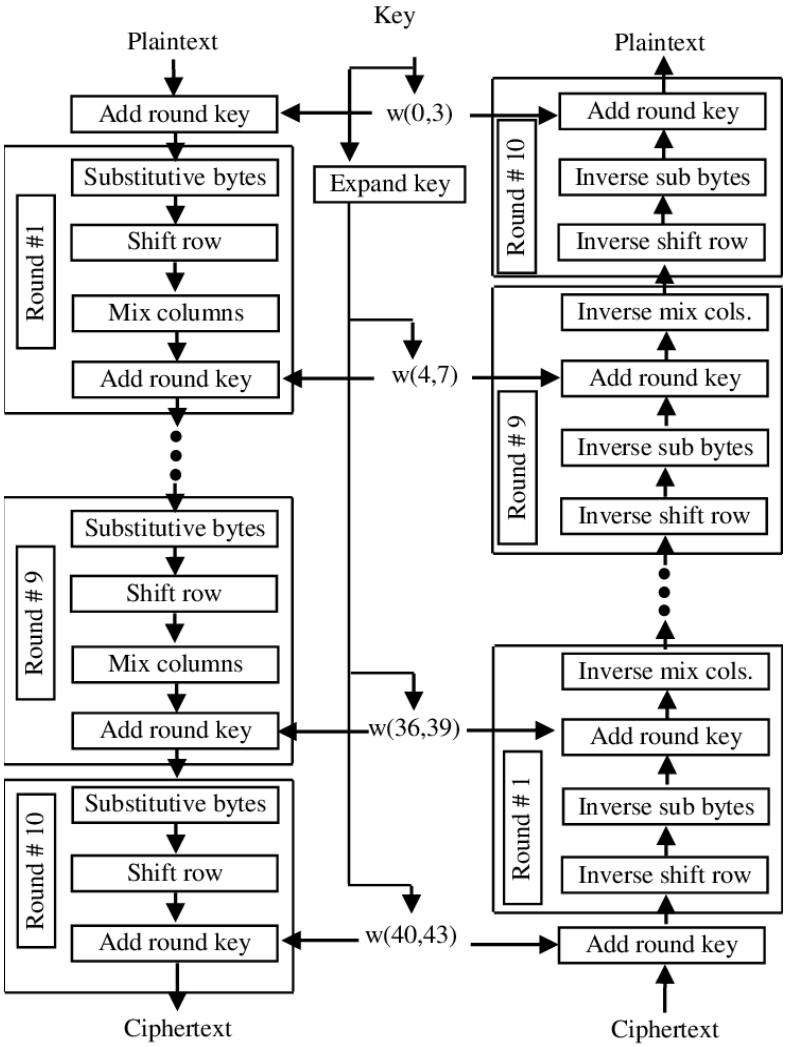

Figura 1: Diagrama general de AES (Source: Study of WiMAX Based Communication Channel Effects on the Ciphered Image Using MAES Algorithm)

AES soporta tamaños de clave de 128, 192 y 256 bits, ofreciendo flexibilidad para diferentes necesidades de seguridad. Además, utiliza una estructura de red de sustitución-permutación (*Substitution-Permutation Network*, SPN), a diferencia de la red de Feistel [62] utilizada en

DES, lo que permite que sea eficiente tanto en implementaciones software como hardware y haciéndolo ideal para un proyecto como este. La estructura SPN en AES consiste en una serie de operaciones matemáticas que se aplican repetidamente en cada ronda. Esta estructura proporciona una fuerte difusión y confusión, elementos esenciales para la seguridad criptográfica. Sin embargo, la red de Feistel divide el bloque de datos en dos mitades y opera en una mitad en cada ronda, utilizando una función F y operaciones XOR. Aunque las mencionadas redes de Feistel tienen la ventaja de que el cifrado y descifrado son estructuralmente idénticos, la estructura SPN de AES permite una mayor paralelización y eficiencia, especialmente en implementaciones hardware.

La información se procesa en bloques de 16 bytes, representados como una matriz de 4x4, conocida como "*State Array*". Cada celda de esta matriz contiene un byte de la información total. Es importante destacar que AES opera con datos a nivel de bytes, no de bits, tratando así el bloque de 128 bits como 16 bytes. Para ilustrar mejor esta organización de datos, en la Figura 2 se muestra la transformación de un bloque lineal de 16 bytes (numerados del 0 al 15) a su representación matricial 4x4, donde se puede observar claramente cómo los bytes se distribuyen en la matriz siguiendo un orden de izquierda a derecha y de arriba abajo.

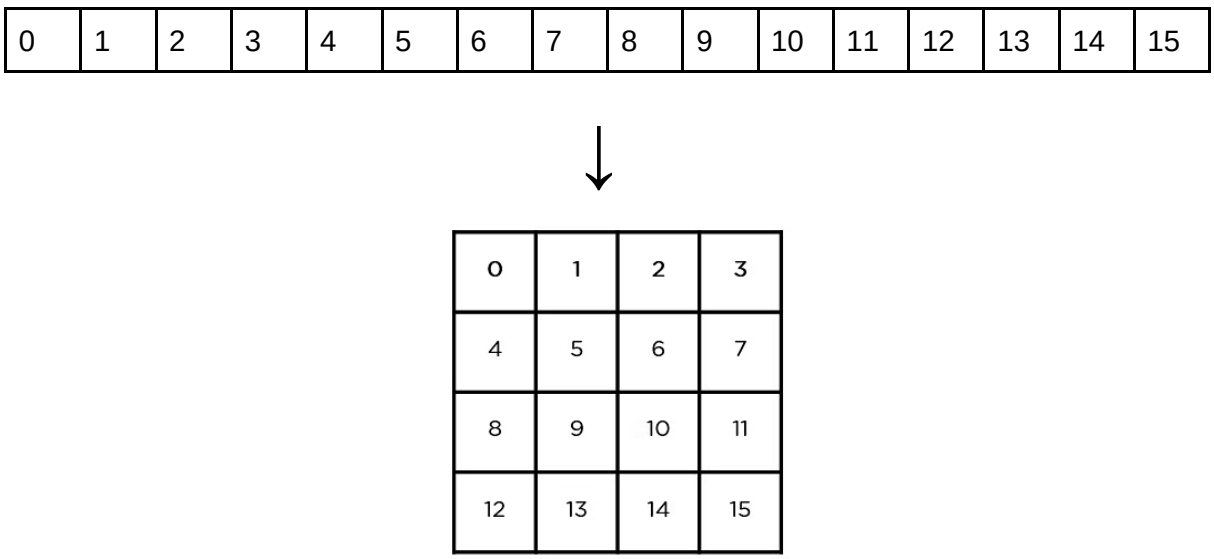

Figura 2: Matriz State Array

El algoritmo se compone de una serie de rondas que transforman el texto plano en texto cifrado. El número de rondas depende del tamaño de las claves: 10 rondas para claves de 128 bits, 12 para 192 bits, y finalmente, 14 para 256 bits.

Antes de describir los pasos básicos para el encriptado, es crucial comprender las operaciones básicas que lo componen. Estas operaciones están diseñadas para proporcionar dos propiedades criptográficas fundamentales:

- La confusión se refiere a hacer la relación entre la clave y el texto cifrado lo más compleja posible. Esto se logra principalmente a través de la operación SubBytes, que implementa una sustitución no lineal.

- La difusión, por otro lado, busca que cada bit del texto plano influya en muchos bits del texto cifrado. Esto se consigue principalmente a través de las operaciones ShiftRows y MixColumns, que dispersan la información a través del State Array. ShiftRows realiza un desplazamiento cíclico de las filas (cada fila se desplaza un número diferente de posiciones hacia la izquierda), mientras que MixColumns aplica una transformación matemática a cada columna mediante multiplicaciones en un campo finito, mezclando así los bytes dentro de cada columna. Estas dos operaciones en conjunto garantizan que cualquier cambio en un byte se propague a través de toda la estructura.

Estas dos propiedades, trabajando en conjunto, aseguran que pequeños cambios en el texto plano o en la clave resulten en cambios significativos y aparentemente aleatorios en el texto cifrado, lo que hace extremadamente difícil para un atacante deducir el texto plano o la clave a partir del texto cifrado.

Las operaciones básicas de AES son:

- AddRoundKey (Suma de la clave de la ronda): En esta operación, el State Array se combina con la subclave correspondiente a la ronda actual mediante una operación XOR bit a bit. mediante una operación XOR bit a bit. Esto aporta la dependencia de la clave al proceso de cifrado.

- SubBytes (Sustitución de bytes): En este paso se sustituye cada byte del State Array por otro según una tabla predefinida llamada S-box. Su objetivo es crear confusión al cifrado. La S-box se construye utilizando la función inversa en el campo finito GF(2^8), seguida de una transformación afín. Esta construcción asegura que la S-box tenga propiedades criptográficas fuertes, como alta no linealidad y baja correlación. S-box es importante debido a su capacidad para proporcionar confusión, rompiendo cualquier patrón lineal en los datos. Como se muestra en la Figura 3, la tabla S-box representa todos los posibles valores de sustitución en formato hexadecimal, organizados en una matriz de 16x16 para su consulta.

		Y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
X	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	a	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	b	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	c	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	d	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	e	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	f	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figura 3: Tabla S-box

- ShiftRows (Desplazamiento de filas): En este paso los bytes de cada fila del State Array se desplazan cíclicamente. Forma parte de la permutación de SPN. Contribuye a la difusión del algoritmo. La primera fila no se desplaza, la segunda fila se desplaza 1 byte a la izquierda, la tercera fila 2 bytes, y la cuarta fila 3 bytes. Este desplazamiento asegura que los bytes de una columna se dispersen a diferentes columnas en las siguientes rondas.

- MixColumns: Esta transformación lineal opera en cada columna del State Array, multiplicándose por una matriz constante. Esto aumenta aún más la difusión del algoritmo. Utiliza multiplicación en el campo finito GF(2^8). Cada byte de una columna se reemplaza por una función de todos los bytes de la misma columna. Matemáticamente, esto se realiza multiplicando cada columna por una matriz fija, como podemos ver representado en la Figura 4.

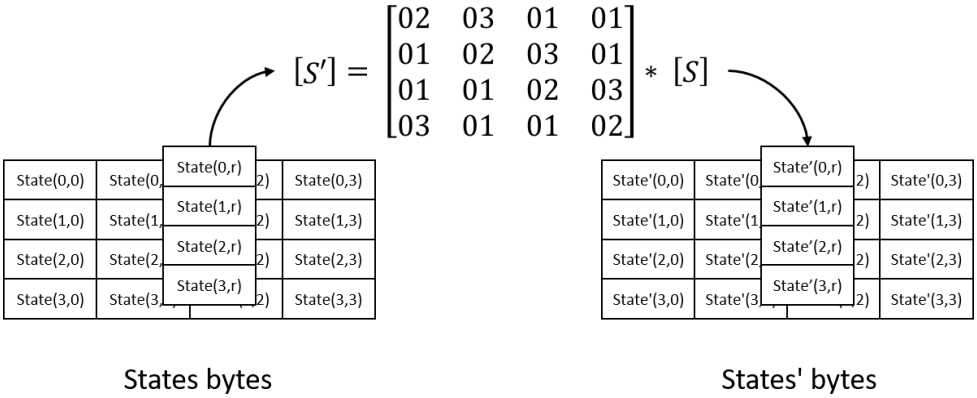

Figura 4: Diagrama de la operación MixColumns (Source:Avalanche Effect of AES Algorithm)

La multiplicación en GF(2^8) se realiza módulo el polinomio irreducible $x^8 + x^4 + x^3 + x + 1$. Esta operación asegura que cada byte de salida dependa de todos los bytes de entrada de la columna, proporcionando una fuerte difusión.

La estructura básica de AES consta de los siguientes pasos:

- Expansión de la clave: Se expandió la clave original para obtener las subclaves necesarias para cada ronda. Dicha clave se expande a $n + 1$ *keys* (siendo n el número de rondas). En AES 128 esto equivaldría a 10 rondas, utilizando 11 llaves. A este proceso se le conoce como Rijndael Key Schedule. El proceso de expansión de clave utiliza una combinación de operaciones que incluyen rotaciones de bytes, sustituciones usando la S-box, y XOR con constantes de ronda. Este proceso asegura que cada subclave sea suficientemente diferente de las demás y del material de clave original, aumentando la seguridad del algoritmo.

- Ronda inicial: Se realiza una operación AddRoundKey, donde el State Array se combina con la clave inicial mediante una operación XOR bit a bit.

- Rondas principales: Cada ronda consiste en las cuatro operaciones anteriormente mencionadas: SubBytes, ShiftRows, MixColumns y AddRoundKey.

- Ronda final: Se realiza de la misma forma que las anteriores, pero omitiendo la operación MixColumns.

Es importante señalar que el proceso de descifrado en AES utiliza las operaciones inversas de las mencionadas anteriormente, aplicadas en orden inverso. Esto implica el uso de InvSubBytes, InvShiftRows e InvMixColumns, junto con AddRoundKey. Se comienza con la última subclave y se trabaja hacia atrás.

Se puede implementar eficientemente utilizando tablas de búsqueda precalculadas para las operaciones SubBytes y MixColumns.

Para ilustrar el funcionamiento práctico del algoritmo AES, se ha desarrollado un ejemplo detallado de su implementación que se encuentra documentado en el Apéndice A.

En los últimos años, se ha introducido una mejora significativa en el rendimiento de AES: Instrucciones Nuevas del Estándar de Cifrado Avanzado (*Advanced Encryption Standard New Instructions,* AES-NI) [63]. Consiste en un conjunto de instrucciones específicas para AES integradas en el hardware de muchos procesadores modernos, introducido por Intel en 2010 y posteriormente adoptado por AMD. Estas instrucciones están diseñadas para acelerar significativamente las operaciones clave de AES.

5.3 Modos de operación

Como ya se ha explicado en el punto anterior, AES es un cifrado en bloques de tamaño fijo normalmente de 16 bytes (128 bits). Sin embargo, la manera en la que se procesan estos datos varía dependiendo del modo de operación que se use. Estos modos de operación determinan cómo se van a realizar las operaciones entre los distintos bloques. Este trabajo se centrará en tres modos concretos: ECB, CBC y CTR [64].

5.3.1 ECB (*Electronic Codebook*)

ECB es el modo de operación más simple, pero también el más inseguro. Consiste en dividir el *plaintext* (texto plano) en bloques iguales, rellenando con *padding* (bytes adicionales que se añaden cuando el último bloque no alcanza el tamaño requerido) en caso de ser necesario. Cada bloque se cifra de forma independiente utilizando la misma clave, sin ninguna relación con los otros bloques. El proceso de desencriptado es similar, cada bloque cifrado se desencripta de forma independiente usando la misma clave. En la Figura 5 se presenta un diagrama que ilustra este proceso secuencial, donde cada bloque de texto plano (P1, P2, P3) se cifra independientemente para producir los bloques cifrados correspondientes (C1, C2, C3).

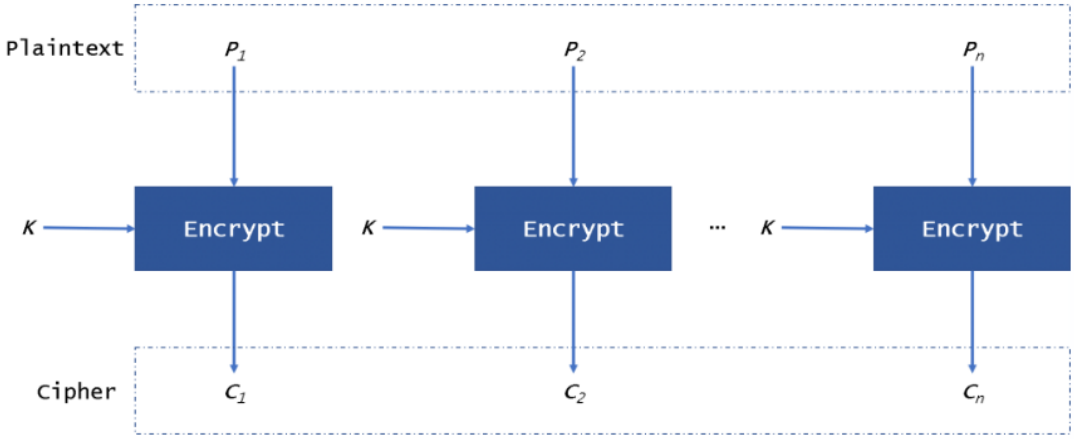

Figura 5: Diagrama de cifrado AES (Source: The difference in five modes in the AES encryption algorithm)

ECB es el modo más rápido, fácil de implementar y de entender. Además, no requiere de IV (concepto que se explicará y aplicará en otros modos). Este modo además es robusto ante la propagación de errores, debido a que un fallo en un bloque no causará problemas en los siguientes.

Uno de los mayores problemas que tiene este modo surge a raíz de que dos bloques iguales darán lugar al mismo encriptado. Aunque esto sí que permite la paralelización y por tanto aceleración en ciertas implementaciones hardware, sin embargo, provoca que se observen patrones que son vulnerables a ataques. Esta es una de sus debilidades más fáciles de explotar. Por otro lado, como los bloques son independientes un atacante, Mallory (*malicious*), podrá reordenar o borrar bloques sin que se le detecte, causando así confusión y errores en la comunicación. La ilustración siguiente muestra visualmente este problema de patrones, donde se puede observar cómo el cifrado ECB mantiene reconocible la estructura de la imagen original.

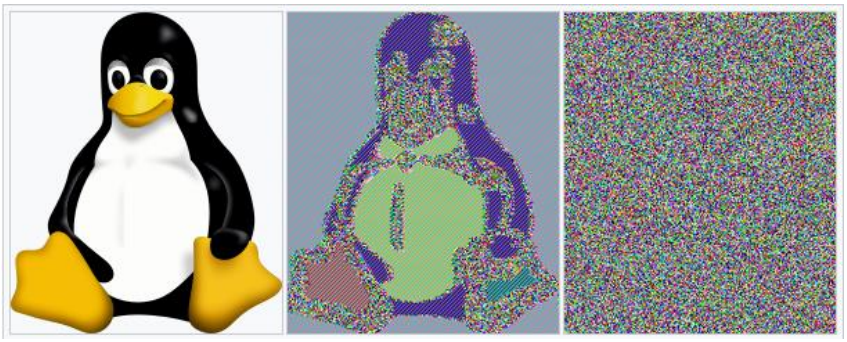

Figura 6: Comparación de cifrado ECB contra otros modos (Source: Un Análisis Profundo de los Modos de Cifrado: ECB, CBC)

ECB utiliza *padding* (relleno) para asegurar que el tamaño del *plaintext* sea un múltiplo del tamaño del bloque. Este *padding* puede ser vulnerable a ciertos ataques como el *Padding Oracle Attack*, donde un atacante puede manipularlo para obtener información sobre el texto original.

A pesar de tener grandes problemas de seguridad, ECB se sigue utilizando en algún que otro caso, aunque a día de hoy está bastante obsoleto. Dicho modo se puede seguir usando para encriptar pequeños trozos de información donde se sabe con seguridad que no hay repeticiones y por tanto no se podrán discernir patrones.

5.3.2 CBC (*Cypher Block Chaining)*

En CBC, cada bloque del texto plano se combina mediante la operación XOR con el bloque de texto cifrado anterior. Esto quiere decir que los bloques siguientes dependen del bloque anterior, por ello, este modo a diferencia del modo ECB no tiene protección contra la propagación de errores, los errores serán arrastrados a bloques consecutivos. Sin embargo, dos bloques cifrados a partir de un mismo *plaintext* dará resultados diferentes, lo cual soluciona el problema de los patrones y las

vulnerabilidades asociadas a ellos. La Figura 7 ilustra tanto el proceso de cifrado como el de descifrado en CBC, mostrando la dependencia entre bloques mediante las operaciones XOR.

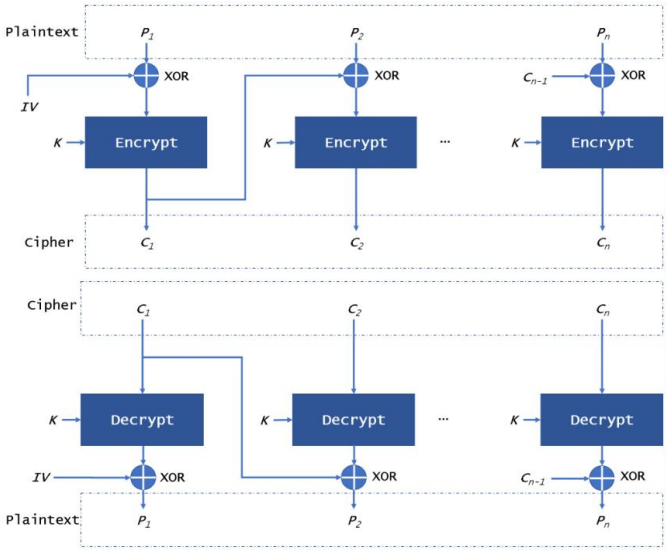

Figura 7: Diagrama de cifrado de bloques CBC (Source: The difference in five modes in the AES encryption algorithm)

Este modo además introduce un concepto nuevo, el vector de inicialización (IV), que se utiliza al principio para realizar la operación XOR con el texto plano. Es crucial que dicho vector sea aleatorio y suficientemente seguro, debido a que existen maneras de explotar y atacar un encriptado a través de él. Además, es recomendable que se utilice un vector de inicialización o IV diferente en cada cifrado para evitar la creación de patrones. El IV no necesita ser secreto, pero debe ser aleatorio.

Cabe destacar que el modo CBC tiene las mismas necesidades y problemas relacionados con el *padding* que tiene el modo ECB. Además, al utilizar XOR, es vulnerable a ataques de *Bit-Flipping*, donde si el atacante invierte ciertos bits puede causar cambios controlados en el texto descifrado, causando confusión y un cambio de mensaje.

Se podría decir que CBC es el modo más utilizado actualmente, debido a que aumenta la seguridad considerablemente en comparación con ECB al proporcionar confidencialidad y difusión del texto plano. Se utiliza por ejemplo para almacenamiento de datos y comunicaciones seguras. Una gran ventaja, es que, aunque el encriptado no se puede paralelizar, el desencriptado si, convirtiéndolo en uno de los más rápidos

en hacerlo. Sin embargo, CBC no es adecuado para el cifrado de datos en tiempo real por su naturaleza secuencial.

5.3.3 CTR (*Counter*)

El modo CTR utiliza en vez de un IV, un *counter* (contador), que funciona de manera similar. Este *counter* se inicializa con un número que se utilizará una única vez llamado *nonce* y será incrementado en cada bloque. Esto proporciona aleatoriedad y que cada texto cifrado sea único, aunque se use la misma clave múltiples veces. La principal diferencia respecto a CBC, es que en el modo CTR, el *nonce* es usado como parte de la entrada al bloque de cifrado para generar las claves. Una característica particular de este modo es que el desencriptado se realiza de la misma forma que el encriptado, como se ilustra en la figura a continuación.

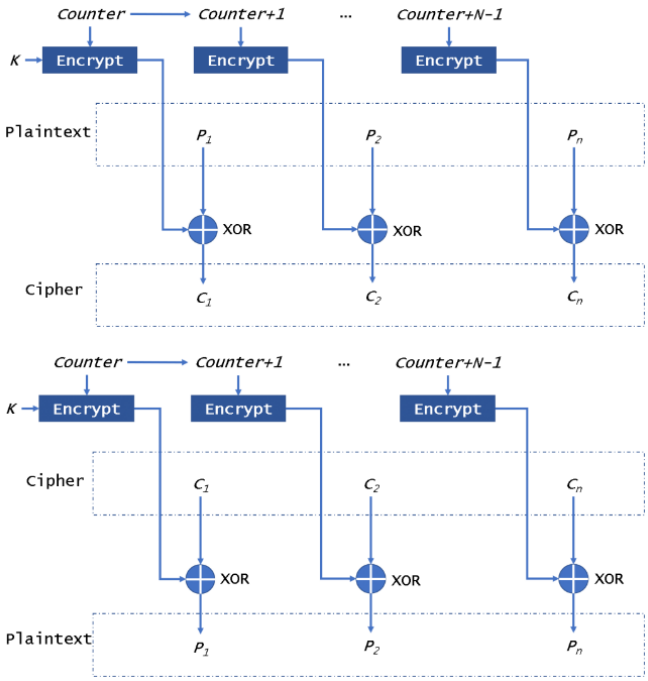

Figura 8: Diagrama de cifrado de bloques CTR (Source: The difference in five modes in the AES encryption algorithm)

Teniendo en cuenta que cada bloque se procesa de manera independiente, ocurre lo mismo que en ECB en términos de la paralelización. Tanto su cifrado como descifrado se puede paralelizar convirtiéndolo en un modo muy rápido. Del mismo modo, CTR no comete propagación de errores. Además, CTR a diferencia de los modos anteriores, no requiere de *padding* lo cual lo hace más seguro contra ataques del mismo.

Como en el caso del IV, es crucial que el *nonce* no se reuse y además sea suficientemente único y aleatorio. Al utilizar operaciones XOR es también vulnerable a ataques de inversión de bits.

CTR es bastante usado a día de hoy debido a su eficiencia, velocidad y capacidad de paralelización. Se observa su uso en protocolos de comunicación como TLS (*Transport Layer Security,* Seguridad de la Capa de Transporte) [65] y IPsec (*Internet Protocol Security,* Seguridad del Protocolo de Internet) [66], del mismo modo se utiliza en inscripciones de disco para proteger datos. Además, CTR se beneficia de implementación hardware que utiliza sus características de aceleración de procesadores modernos para incrementar significativamente su velocidad, del mismo modo, procesadores multicore pueden aprovechar su paralelización.

5.3.4 Otros modos no autenticados (CFB, OFB)

Hasta ahora se ha estado viendo modos no autenticados. Esta categorización se refiere a si un modo incluye o no mecanismos para asegurar tanto la confidencialidad como la integridad en el encriptado de datos. Los modos no autenticados se centran principalmente en la confidencialidad, sin abordar la integridad, a diferencia de los modos autenticados que se explican en el siguiente punto, los cuales trabajan ambos aspectos. Algunos de los modos no autenticados más importantes ya se han descrito, a continuación, se mencionan otros dos de interés.

CFB (*Cipher Feedback,* Retroalimentación de Cifrado) [67] y OFB (*Output Feedback*, Retroalimentación de Salida) [68] son modos de operación para cifrados de bloque. CFB convierte el cifrado de bloque en un cifrado de flujo, cifrando la salida del bloque anterior para generar un flujo de claves que se combina con el texto plano. Este modo es particularmente útil en situaciones donde los datos llegan en un flujo continuo (como en comunicaciones de red en tiempo real). Algo destacable de CFB es que los errores se auto-sincronizan después de un número de bloques, lo que lo hace resistente a ciertos tipos de errores de transmisión. Por otro lado, CFB genera un flujo de claves cifrando repetidamente un vector de inicialización, lo que permite pre-calcular el flujo de claves antes de conocer el texto plano. Esta característica lo hace adecuado para cifrar datos de *streaming*, como transmisiones de vídeo o audio. Además, OFB no propaga errores, lo que puede ser beneficioso en entornos con alta probabilidad de errores de transmisión.

Aunque menos comunes que los inicialmente citados ECB, CBC o CTR, estos modos tienen aplicaciones específicas en ciertos escenarios de seguridad. CFB se utiliza a menudo en protocolos de red donde la sincronización es crucial, mientras que OFB encuentra aplicación en sistemas que requieren pre-procesamiento del flujo de claves.

5.3.5 Modos autenticados (GCM y CCM)

Los modos autenticados son los más empleados en el mundo real debido a su capacidad para proporcionar tanto confidencialidad como integridad. Para lograr esta autenticación, se suelen combinar modos tradicionales no autenticados con mecanismos adicionales que se encargan de detectar si los datos encriptados han sido modificados o no. Los modos más comunes son GCM (*Galois/Counter Mode*) [69] y CCM (*Counter with CBC-MAC*) [70].

GCM es uno de los modos más usados en la actualidad y con razón. Combina el modo Counter (CTR) con mecanismos de autenticación de Galois (GHASH) que utiliza para calcular una etiqueta de autenticación sobre el texto cifrado. Es altamente eficiente y permite la paralelización, lo que lo hace ideal para aplicaciones de alto rendimiento.

Por otro lado, CCM (Counter con CBC-MAC) utiliza el modo CBC pero genera un mensaje autenticado que cubre todo el *plaintext* (no solo el texto cifrado), asegurando mantener la integridad en todo el proceso.

CCM es menos eficiente que GCM pero más simple de implementar.

6. Implementación de AES

En esta sección se detalla la implementación de AES-128, en tres modos de operación diferentes previamente descritos y que son: ECB, CBC y CTR. La implementación se ha realizado en Python puro sin ningún tipo de librería criptográfica externa, debido a dos razones principalmente, como ya se mencionó. Por un lado, se pretendía entender completamente el funcionamiento del algoritmo implementado desde cero. Por otro lado, como más adelante se necesitaba transformar el código en VHDL para la implementación en la FPGA, tener un código sin librerías externas o dependencias hacía esta transformación más fácil.

6.1 Detalles generales de implementación

La implementación sigue un diseño modular, donde se han separado los diferentes componentes del algoritmo AES en funciones independientes. Esta estructura mejora la lectura del código para su posterior transformación y mantenibilidad. Es importante mencionar que las operaciones en AES se realizan bit a bit, en concreto las operaciones previamente mencionadas: SubBytes, ShiftRows, MixColumns y AddRoundKey. Para manejar estas operaciones de manera eficiente en Python, se han utilizado estructuras de datos como bytearray para el *state,* lo que permite realizar manipulaciones a nivel de bit de manera efectiva.

Se han tenido en cuenta varias consideraciones:

- *Padding*: Se ha implementado el método PKCS7 [71], que es ampliamente utilizado y compatible con el estándar AES. Este método consiste en añadir bytes al final del bloque, donde el valor de cada byte añadido es igual al número total de bytes de relleno necesarios. Esta característica lo hace único y fácilmente reversible, diferenciándolo de otros métodos de *padding* más simples.

- Manejo de errores: El código implementa un manejo robusto de errores a lo largo de todas las funciones principales, asegurando la integridad y la seguridad durante el proceso de cifrado y descifrado de AES. Se utilizan excepciones estándar de Python, como `ValueError` y `TypeError`, para capturar y gestionar situaciones específicas.

- Formateo: Se ha utilizado el formato *Big Endian* [72] debido a que es el estándar más usado en implementaciones de AES y en el NIST. Por otro lado, se ha utilizado

representación hexadecimal tanto para el texto en plano como para las llaves. Esta decisión viene dada porque es un formato muy compacto que ayuda a la lectura y, por otro lado, tiene una conversión fácil a binario que es esencial para las operaciones bit a bit que se utilizan.

- Tamaño de la clave: La implementación se limita a claves de 128 bits y bloques de datos de 128 bits también, lo que corresponde a AES-128. No se han implementado variantes de AES con tamaños de clave y bloques diferentes (AES-192 o AES-256).
- Vectores test: Para probar el código, se ha utilizado vectores de prueba del NIST, específicamente los documentos NIST SP 800-38A y AESAVS [73], lo que asegura la conformidad con los estándares oficiales.
- Modos de operación: Se quería probar varios modos de operación, aunque algunos no fueran los más seguros, para compararlos entre sí.

Además, es importante mencionar que la implementación tiene algunas limitaciones y problemas conocidos:

- Generación de números aleatorios: Para la generación de números aleatorios en las pruebas, se utiliza *secrets.token_bytes()* [74], que es más seguro que la biblioteca *random* estándar de Python. Esta elección se debe a que *secrets.token_bytes()* no depende de una semilla predeterminada, sino que obtiene la entropía directamente del sistema operativo, evitando así la predictibilidad que puede surgir al usar semillas en generadores pseudoaleatorios tradicionales. No obstante, para una implementación en producción, se podrían considerar fuentes de entropía adicionales para mayor robustez, aspecto que se detallará posteriormente
- Rendimiento: aunque el código está implementado en Python puro sin librerías externas, se han realizado algunas optimizaciones, como el uso de tablas precalculadas para la multiplicación en el campo de Galois, lo que mejora la eficiencia de operaciones como MixColumns. Sin embargo, el rendimiento aún no es comparable con implementaciones altamente optimizadas y estandarizada-
- Uso de memoria: La implementación mantiene todo en memoria, lo que limita el tamaño de los bloques que se pueden procesar.
- IV: Para modos que requieren IV (como CTR y CBC), es crucial una generación y manejo seguro que no se han implementado.

- Manejo de la clave: La implementación actual no contiene un manejo robusto de la clave. En un escenario real, habría que tener consideraciones extras en cuenta.

A pesar de estas limitaciones, la implementación cumple con los estándares AES-128 y proporciona una base sólida para comprender y estudiar el algoritmo.

Algunos percances que surgieron:

- *Byte ordering:* Inicialmente hubo un poco de confusión con la representación de los bytes, pues no se tuvo en cuenta, y causó resultados incorrectos. Sin embargo, tenía fácil solución: todo se estandarizó, como ya se ha mencionado, usando el formato *Big Endian*.

- Implementación en Galois: La implementación de las operaciones para MixColumns requirió un estudio en profundidad del campo de Galois y sus propiedades matemáticas, ya que este componente es fundamental para el funcionamiento correcto del algoritmo AES.

- Debugging de datos cifrados: Cuando el código daba errores, era bastante difícil de depurar debido a que era complicado encontrar el punto concreto donde fallaba por la naturaleza de los datos encriptados.

6.2 Explicación del código

A partir de la estructura básica de AES explicada anteriormente, esta sección se centra en los detalles específicos de implementación del algoritmo AES-128, destacando los componentes clave y los tres modos implementados: ECB, CBC y CTR.

Implementación de las operaciones principales:

- sustituir_bytes y sustituir_bytes_inverso: Para estas operaciones se han utilizado tablas de sustitución precalculadas. La S-box se define como una lista global de valores hexadecimales para mayor eficiencia, mientras que invSbox se genera dinámicamente a partir de la S-box al inicio del programa (inv_sbox). Esta decisión permite un acceso rápido a ambas operaciones sin necesidad de almacenar dos tablas estáticas.

- Desplazar_filas y desplazar_filas_inverso: Estas funciones se implementan utilizando la notación de corte (llamada *slicing* en inglés) de Python. Esta elección se tomó

porque permite una manipulación eficiente de las filas del estado AES, aprovechando las optimizaciones internas de Python para operaciones con listas.

- Mezclar_columnas y mezclar_columnas_inverso: Estas dos operaciones fueron uno de los mayores desafíos debido a que operan en el campo de Galois (Gmul). Este campo se define sobre polinomios de grado menor que 8 con coeficientes en GF(2) (es decir '1' ó '0'), módulo el polinomio irreducible $x^8 + x^4 + x^3 + x + 1$ utilizado en AES. Originalmente se optó por utilizar una función de multiplicación gmul para aplicarla según la matriz de transformación AES. No es la aproximación más rápida, pero ofrecía buen equilibrio entre legibilidad y eficiencia, además de que proporciona una representación clara del proceso matemático subyacente. Tras comprender correctamente el proceso se decidió optimizar e implementar una función alternativa que utiliza tablas de *lookup* (búsqueda) precalculadas para las multiplicaciones más comunes en el campo de Galois, lo que mejoró significativamente el rendimiento de estas operaciones.

- expandir_clave: Esta función se implementó siguiendo el estándar AES, utilizando una lista Python para almacenar la clave expandida. Esta decisión acelera el acceso a las subclaves durante las rondas de cifrado y descifrado, mejorando el rendimiento general. Además, se utilizó el array rcon [0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1B, 0x36] como constante para la generación de las subclaves, siguiendo las especificaciones del algoritmo AES.

Implementación de los modos de operación y sus particularidades:

- La implementación ECB es la más directa de las tres y no requiere un vector de inicialización (IV), lo que la distingue de los otros modos. Se utiliza la función *encrypt* siguiendo los pasos de AES, y para desencriptar se realizan los pasos en orden inverso utilizando las funciones inv que se han explicado antes. Esta implementación sirvió como base para los otros modos.

- La implementación de CBC requirió de más consideración debido a la dependencia entre bloques que lo caracteriza. La función de cifrado acepta el IV como un parámetro separado, permitiendo flexibilidad en su generación y uso. Aunque no se implemente una generación segura de IV en este código, esta estructura nos permite fácilmente integrar tal funcionalidad en el futuro. Para la implementación del encadenamiento de bloques se optó por un enfoque iterativo, manteniendo una variable que guarda el bloque previo.

En el proceso de descifrado, se presenta la necesidad de asegurarse de preservar el bloque cifrado original antes de descifrarlo ya que se necesita para la operación XOR con el siguiente bloque descifrado. Esto se consiguió con un bucle y variables temporales.

- La principal diferencia entre CTR y los otros modos radica en que este transforma efectivamente el cifrado por bloques en un cifrado de flujo. Un aspecto crucial de la implementación fue la gestión del contador. Se optó por utilizar un *nonce* de 8 bytes combinando con un contador de 8 bytes, formando el bloque de 16 bytes necesario para AES. Esta decisión permitió mantener un equilibrio entre la unicidad del contador y la cantidad de datos que se pueden cifrar de manera segura con una sola clave. A diferencia de CBC, la operación XOR entre el texto plano y el bloque de cifrado se trabajó byte por byte. La naturaleza de CTR como cifrado de flujo permitió implementar un cifrado paralelizable.

6.3 Pruebas, validación y análisis comparativo

Una vez terminada la implementación de AES en sus modos correspondientes, el siguiente paso fue realizar una validación del código y analizar su seguridad y rendimiento con el fin de seleccionar el modo más adecuado para una futura implementación en la FPGA. En los primeros apartados, se explican detalladamente las pruebas realizadas para evaluar los diferentes modos de AES. Posteriormente, se presentan los resultados obtenidos y se comparan con la documentación disponible para verificar su corrección y eficiencia.

6.3.1 Vectores de prueba y verificación

Para asegurar que la implementación de software realizada hacía el cifrado y descifrado de manera correcta se utilizó una serie de vectores de prueba estándar, implementados en la función ejecutar_vectores_prueba(), cuyos resultados se pueden ver en las figuras inferiores. Estos vectores incluyen casos de prueba asegurando que la implementación cumple con los estándares AES en todos los casos. A continuación, se muestra un ejemplo por modo (se realizaron pruebas con *padding* también).

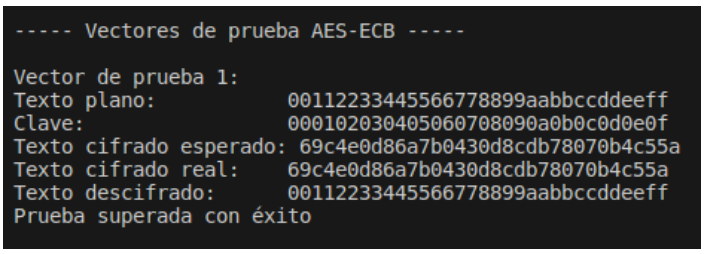

Figura 9: Prueba de verificación en modo ECB

```
----- Vectores de prueba AES-CBC -----

Vector de prueba 1:
Texto plano:           6bc1bee22e409f96e93d7e117393172a
Clave:                 2b7e151628aed2a6abf7158809cf4f3c
IV:                    000102030405060708090a0b0c0d0e0f
Texto cifrado esperado: 7649abac8119b246cee98e9b12e9197d
Texto descifrado:      6bc1bee22e409f96e93d7e117393172a
Prueba superada con éxito
```

Figura 10: Prueba de verificación en modo CBC

```
----- Vectores de prueba AES-CTR -----

Vector de prueba 1:
Texto plano:           6bc1bee22e409f96e93d7e117393172a
Clave:                 2b7e151628aed2a6abf7158809cf4f3c
Nonce:                 f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff
Texto cifrado esperado: 874d6191b620e3261bef6864990db6ce
Texto cifrado real:    874d6191b620e3261bef6864990db6ce
Texto descifrado:      6bc1bee22e409f96e93d7e117393172a
Prueba superada con éxito
```

Figura 11: Prueba de verificación en modo CTR

Estos tests proporcionaron validación de la corrección, asegurando que se cumple con el estándar AES y que la identificación de errores fue implementada correctamente. Sin embargo, estos tests sólo comprueban su corrección y no cosas como su seguridad o rendimiento, por ello, se desarrollaron más tests que se detallan en la siguiente sección.

6.3.2 Pruebas de rendimiento

Para evaluar el rendimiento de la implementación, se desarrolló la función: probar_tiempo_ejecución():

Esta prueba mide el tiempo en segundos que tarda el algoritmo en cifrar y descifrar bloques de diferentes tamaños (16, 128 y 1024 bytes). Se realizan 500 iteraciones para cada tamaño para obtener un promedio confiable.

Con esta prueba también se averiguó la latencia, que se define como el tiempo que tarda en procesarse un solo bloque de datos. La medición se realizó tomando el tiempo de ejecución para el bloque de menor tamaño posible (128 bits).

6.3.3 Pruebas de seguridad

Para evaluar la seguridad de la implementación, se diseñó una serie de pruebas específicas destinadas a detectar vulnerabilidades comunes en algoritmos de cifrado:

a) Prueba de propagación de errores (probar_propagacion_errores()): Esta prueba, específica para CBC, evalúa cómo los errores en el texto cifrado se propagan durante el descifrado.

b) Prueba de difusión (probar_difusion()): Se modifica, un solo bit en el texto plano y se observa cómo esto afecta al texto cifrado. En un cifrado seguro, un cambio en un solo bit del texto plano debería producir cambios en aproximadamente la mitad de los bits del texto cifrado. Esta prueba es crucial para evaluar la propiedad de difusión del algoritmo.

c) Prueba de sensibilidad a la clave (probar_sensibilidad_clave()): Se cifra el mismo texto plano con claves que difieren en un solo bit para verificar que el resultado es significativamente diferente. Esta prueba evalúa la sensibilidad del algoritmo a pequeños cambios en la clave, una propiedad importante para la seguridad del cifrado.

d) Prueba de seguridad del VI (probar_seguridad_iv()): Esta prueba se realizó solo para CBC. Evalúa la importancia de usar IV únicos en dicho modo. Se cifra el mismo mensaje dos veces con IVs diferentes y se comparan los resultados.

e) Prueba de patrón visual (probar_patron_visual()): Esta prueba ayuda a visualizar cómo el algoritmo de cifrado maneja patrones en el texto plano. Es especialmente útil para demostrar la debilidad del modo ECB en preservar patrones del texto original.

f) Prueba de desbordamiento del contador (probar_desbordamiento_contador()): Específica para CTR, verifica el comportamiento del algoritmo cuando el contador alcanza su valor máximo.

Por otro lado, para poder evaluar la calidad de la aleatoriedad producida por los algoritmos, se utilizaron los tests del *NIST Statistical Test Suite* (STS). A diferencia de nuestras otras implementaciones, para estas pruebas sí que se tiene que utilizar una librería especializada debido a la complejidad matemática de los tests, sin embargo, no había ninguna disponible, por lo que se implementaron algunas de las pruebas exprofeso.

STS consta de 15 pruebas estadísticas diseñadas para evaluar la aleatoriedad de secuencias binarias producidas por generadores de números aleatorios o pseudoaleatorios [54].

Algunas de las pruebas más significativas que se realizan en este paquete de tests incluyen:

1. Prueba de frecuencia (monobit): Esta prueba examina la proporción de unos y ceros en la secuencia. Un cifrado seguro debería producir aproximadamente el mismo número de unos y ceros.
2. Prueba de rachas: Analiza la oscilación entre subcadenas consecutivas de unos y ceros. Esta prueba es importante para detectar si hay patrones repetitivos en la salida cifrada.
3. Prueba de complejidad lineal: Determina la complejidad de la secuencia, lo cual es crucial para evaluar la resistencia del cifrado a ataques de criptoanálisis lineal.
4. Prueba de transformada discreta de Fourier: Detecta patrones repetitivos en la secuencia, lo cual podría indicar debilidades en el algoritmo de cifrado.
5. Prueba de entropía aproximada: Compara las frecuencias de bloques superpuestos, lo que ayuda a evaluar la imprevisibilidad de la salida cifrada.

Los resultados de estas pruebas fueron satisfactorios, como se detalla en el siguiente apartado, demostrando que la implementación cumple con las propiedades de seguridad esperadas de AES.

6.3.4 Comparación de resultados

En esta sección, se analizarán y comprarán los resultados obtenidos para las implementaciones de AES en los modos ECB, CBC y CTR. Se contrastan estos resultados con lo que se conoce en la literatura sobre cada modo, para validar la implementación y entender mejor sus características. Todas las pruebas fueron realizadas en un equipo con el siguiente hardware y sistema operativo: Linux, procesador Intel Core i5-7200U de dos núcleos y cuatro hilos, frecuencia base de 2.5 GHz y memoria RAM de 8 GB.

Por claridad, se muestran cómo se han detallado en el punto anterior:

- Rendimiento:
    a) Tiempo de ejecución (en microsegundos):

Modo	16 bytes	128 bytes	1024 bytes
ECB	Encriptar: 0.243 Desencriptar: 2.515	Encriptar: 0.824 Desencriptar: 11.342	Encriptar: 5.579 Desencriptar: 84.733
CBC	Encriptar: 0.238 Desencriptar: 2.693	Encriptar: 0.802 Desencriptar: 12.478	Encriptar: 6.097 Desencriptar: 97.637
CTR	Encriptar: 0.212 Desencriptar: 0.212	Encriptar: 0.988 Desencriptar: 0.988	Encriptar: 5.881 Desencriptar: 5.901
ECB (Crypto)	Encriptar: 0.0038 Desencriptar: 0.0035	Encriptar: 0.0032 Desencriptar:0.0031	Encriptar: 0.0079 Desencriptar: 0.0073
CBC (Crypto)	Encriptar: 0.0057 Desencriptar: 0.0053	Encriptar: 0.0066 Desencriptar: 0.0065	Encriptar: 0.0127 Desencriptar: 0.0141
CTR (Crypto)	Encriptar: 0.0052 Desencriptar: 0.0051	Encriptar: 0.0059 Desencriptar: 0.0059	Encriptar: 0.0108 Desencriptar: 0.0107

Tabla 1: Prueba de rendimiento de AES en Python

Analizando los resultados de la tabla anterior, que recoge la media de tiempos de las 500 ejecuciones realizadas, observamos que los tiempos de ejecución varían entre los diferentes modos:

- ECB: Muestra el mejor rendimiento en términos de velocidad. Esto es consistente con la literatura, ya que ECB es el modo más simple y permite una fácil paralelización.
- CBC: Presenta tiempos de ejecución ligeramente mayores que ECB. Esto se debe a la dependencia entre bloques en el proceso de cifrado.
- CTR: Muestra un rendimiento similar a ECB. Aunque requiere operaciones adicionales para el contador, su capacidad de paralelización compensa el tiempo de ejecución empleado en la realización de estas operaciones extras, resultando buenos tiempos de ejecución. Además, su tiempo de desencriptado es el más bajo de los modos.

b) Latencia y rendimiento:

La latencia como se ha explicado ya, mide el tiempo que tarda en procesarse un solo bloque de datos. Conseguimos los siguientes resultados observados en la tabla anterior:

- ECB: Presenta la menor latencia debido, como siempre, a su simplicidad. Ofrece el mayor *throughput* (rendimiento) debido a su capacidad de paralelización completa.
- CBC: Muestra una latencia mayor en el cifrado debido a la dependencia entre bloques, pero similar a ECB en el descifrado. Además, el *throughput* es menor en el cifrado debido a su naturaleza secuencial, pero comparable a ECB en el descifrado.
- CTR: Ofrece una latencia baja, tanto en cifrado como en descifrado. Presenta un *throughput* alto similar al de ECB.

Estos resultados de latencia son relevantes para aplicaciones que requieren respuestas en tiempo real. Adicionalmente, los resultados de *throughput* son especialmente importantes para aplicaciones que manejan grandes volúmenes de datos.

c) Análisis comparativo:

Los resultados se alinean en gran medida con lo esperado. Schneier (2015) [75] señala que ECB y CTR suelen ser los modos más rápidos debido a su capacidad de paralelización, mientras que CBC es generalmente más lento en el cifrado, pero puede ser más rápido en el descifrado.

Sin embargo, es importante notar que el rendimiento superior de ECB es debido a la seguridad, como se verá en la siguiente sección y como ya se ha mencionado someramente varias veces. CTR ofrece un buen equilibrio, combinando un rendimiento cercano al de ECB con propiedades de seguridad mejores, lo cual lo convierte en el modo predilecto para el proyecto.

CBC, aunque más lento en el cifrado, ofrece propiedades de seguridad robustas y su rendimiento en el descifrado es competitivo. Esto lo hace adecuado para muchas aplicaciones, especialmente aquellas donde la seguridad es prioritaria sobre la velocidad de cifrado.

Estos resultados de rendimiento deben considerarse en conjunto con las propiedades de seguridad de cada modo para tomar decisiones

informadas sobre qué modo es más adecuado para una aplicación específica.

- Seguridad:
  a) Pruebas de propiedades criptográficas:
     ➢ ECB:
        - Difusión: Los resultados mostraron una difusión promedio de 64.19 bits diferentes de 128, lo que representa un 50.15% de cambio. Sin embargo, la difusión está limitada a bloques individuales, sin propagación a bloques subsiguientes. Estos resultados confirman las advertencias de Ferguson et al. (2010) [76] sobre la debilidad de ECB en preservar patrones de texto plano en mensajes más largos.
        - Sensibilidad a la clave: Se observaron 260 bits diferentes de 512 al cambiar la clave, lo que representa un 50.78% de diferencia, indicando una fuerte sensibilidad a cambios en la clave. No obstante, al igual que con la difusión, esta sensibilidad está confinada a bloques individuales sin efecto cascada en los bloques consecutivos debido a la naturaleza del modo ECB.
        - Fuga de información: Los patrones en el texto plano se reflejan directamente en el texto cifrado, especialmente en mensajes más largos o con patrones repetitivos. Esto representa una seria vulnerabilidad en términos de confidencialidad.

Como se puede observar en la Figura 12, las pruebas visuales muestran claramente cómo el texto cifrado mantiene patrones reconocibles del texto original, evidenciando la debilidad principal del modo ECB.

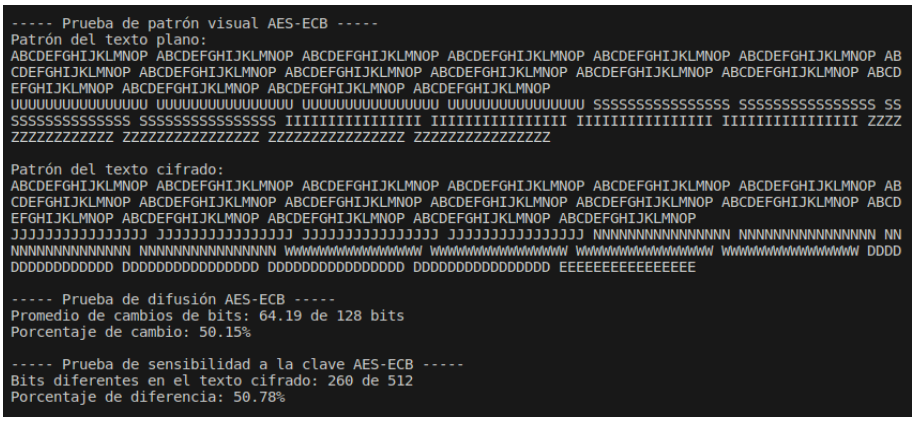

Figura 12: Prueba de seguridad en modo ECB

- CBC:
  - Difusión: Los resultados mostraron una difusión promedio de 160.15 bits diferentes de 384, lo que representa un 41.71% de cambio. Aunque este porcentaje es ligeramente inferior al ideal del 50%, sigue siendo un resultado sólido que demuestra una buena difusión. Los cambios se propagan a bloques subsiguientes, validando la afirmación de Dworkin (2001) sobre la superior difusión de CBC comparada con ECB.
  - Sensibilidad a la clave: Cambios en la clave afectan significativamente el texto cifrado (119 bits diferentes de 256), demostrando una buena sensibilidad a la clave.
  - Prueba IV: Demostró la importancia de usar IVs aleatorios y únicos para cada operación de cifrado. IVs diferentes produjeron cifrados diferentes para el mismo texto plano, mientras que se advirtió correctamente sobre los riesgos de usar IVs predecibles o reutilizados.
  - Propagación de errores. La prueba de propagación de errores demostró claramente cómo un cambio en un solo bloque afecta a todos los bloques subsiguientes.
  - Patrón visual: La prueba de patrón visual mostró una mejora significativa respecto a ECB. Aunque el primer bloque de cada línea muestra un patrón reconocible (probablemente debido al IV), el resto del texto cifrado parece aleatorio y no revela patrones del texto plano. Esto demuestra la efectividad de CBC en ocultar patrones y estructuras del mensaje original.

  En la Figura 13 se presentan los resultados de las pruebas de seguridad en modo CBC, demostrando la mejora en la ocultación de patrones respecto al modo ECB.

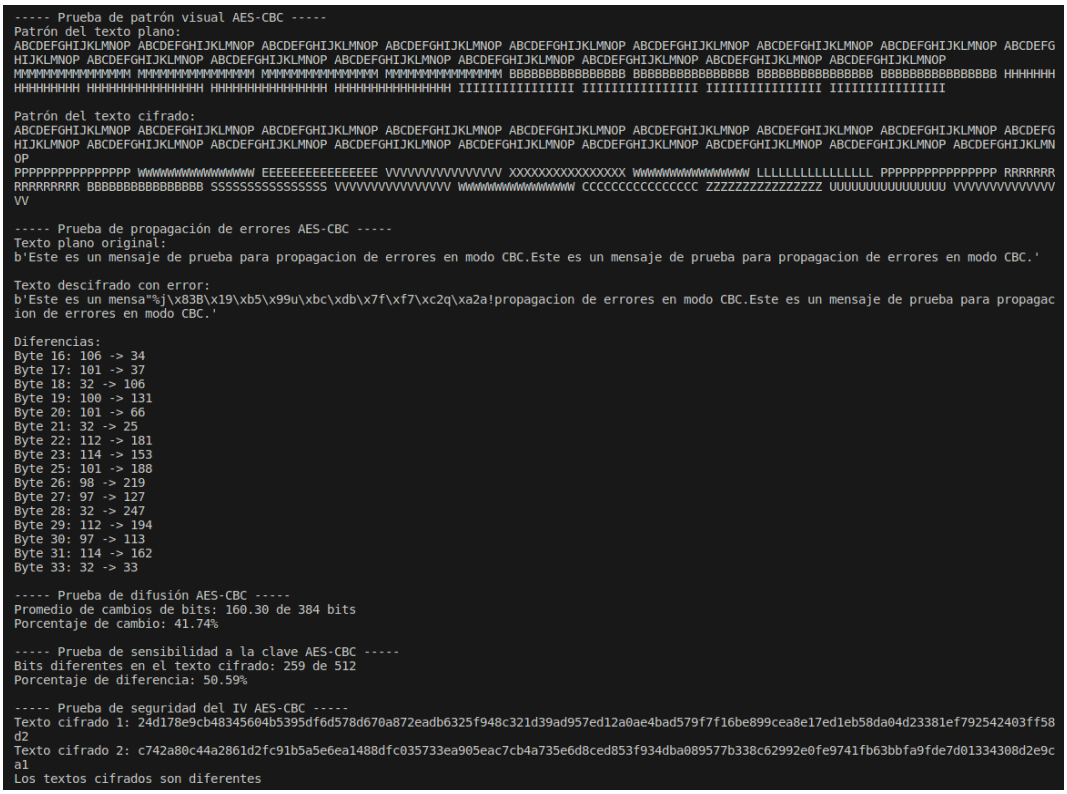

Figura 13: Pruebas de seguridad en modo CBC.

- CTR:
  - Difusión: Los resultados mostraron un promedio de cambios de bits de 1.00 de 256 bits, lo que representa un 0.39% de cambio. Este resultado, aunque puede parecer bajo a primera vista, es exactamente lo esperado y correcto para el modo CTR, que opera de manera similar a un cifrado de flujo. Aunque no hay propagación de cambios entre bloques, esto no compromete la seguridad del modo CTR cuando se implementa correctamente.
  - Sensibilidad a la clave: Se observaron 240 bits diferentes de 456 al cambiar la clave, lo que representa un 52.63% de diferencia. Este resultado es excelente y ligeramente superior al ideal del 50%, indicando una fuerte sensibilidad a cambios en la clave. Esto confirma que los cambios en la clave afectan significativamente a todo el texto cifrado.
  - Seguridad del *nonce*: La prueba del contador demostró que *nonce*s diferentes produjeron cifrados completamente distintos para el mismo texto plano. Esto es crucial para la seguridad del modo CTR, ya que garantiza que cada operación de cifrado es única, incluso si se cifra el mismo mensaje múltiples veces.

- Patrón visual: La prueba de patrón visual mostró una mejora significativa respecto a ECB. Aunque la primera línea muestra un patrón reconocible (probablemente debido al contador inicial), el resto del texto cifrado parece aleatorio y no revela patrones del texto plano.

Como se observa en la Figura 14, las pruebas de seguridad en modo CTR demuestran la aleatorización efectiva del texto cifrado, donde el patrón de caracteres no revela información sobre el texto original.

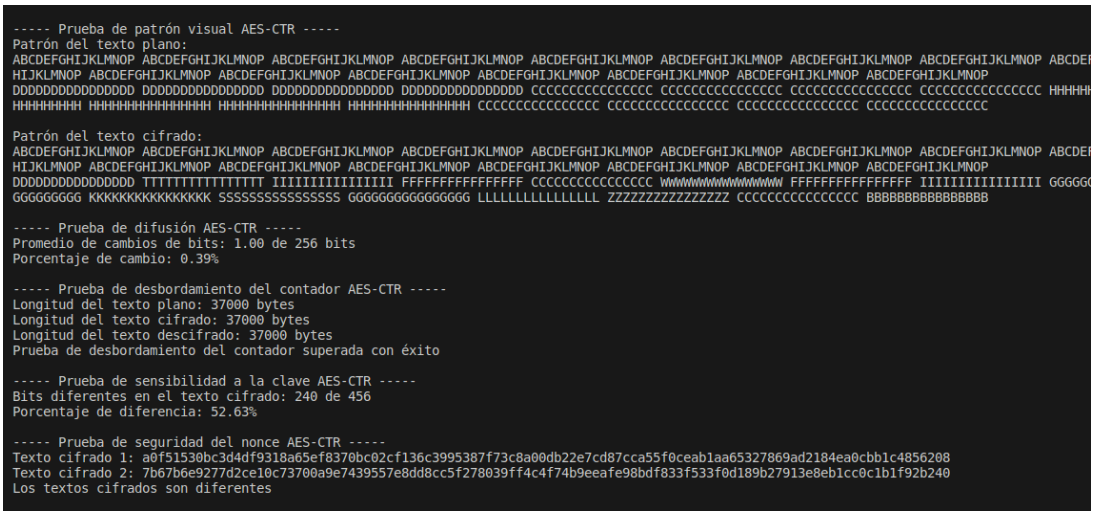

Figura 14: Prueba de seguridad en modo CTR

b) Pruebas NIST STS

Antes de analizar los resultados, es importante entender el concepto de p-valor en el contexto de pruebas de aleatoriedad. El p-valor representa la probabilidad de que una secuencia verdaderamente aleatoria muestre las mismas o peores características de no-aleatoriedad que la secuencia que estamos probando. Un p-valor mayor que el nivel de significancia (típicamente 0.01) sugiere que la secuencia es indistinguible de una verdaderamente aleatoria para ese test específico.

Para este trabajo se seleccionaron las pruebas de Frecuencia, Rachas y Racha Más Larga por su complementariedad en la evaluación de propiedades fundamentales de cifrado: la prueba de Frecuencia evalúa la distribución equilibrada de bits (propiedad básica de todo buen cifrado), mientras que las pruebas de Rachas y Racha Más Larga

analizan la independencia de bits a diferentes escalas, detectando patrones que podrían indicar debilidades en la difusión. Aunque el NIST STS incluye otras pruebas valiosas como la de entropía aproximada o transformada de Fourier, estas tres proporcionan una base sólida para evaluar las propiedades fundamentales de aleatoriedad. En futuras extensiones del trabajo, sería interesante implementar la prueba de complejidad lineal y la de entropía aproximada para un análisis más exhaustivo.

- ECB: Contrario a lo esperado, ECB pasó las pruebas de Frecuencia y Rachas con p-valores aceptables. Sin embargo, falló significativamente en la prueba de Racha Más Larga, lo cual es consistente con las conocidas limitaciones de ECB en términos de seguridad y aleatoriedad. Este resultado mostrado en la figura 15 sugiere que, aunque ECB puede producir salidas que parecen aleatorias en algunas métricas simples, falla en pruebas más rigurosas que detectan patrones a largo plazo.

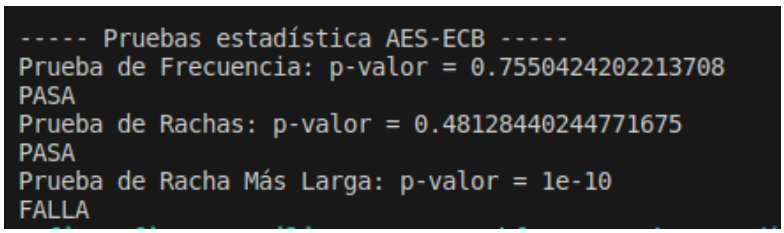

Figura 15: Prueba estadística en modo ECB

- CBC: Mostró una mejora en las pruebas de Frecuencia y Rachas comparado con ECB, con p-valores más altos que indican una mejor aleatoriedad en estas métricas, como se puede observar en la figura inferior. Sin embargo, al igual que ECB, falló en la prueba de Racha Más Larga. Esto sugiere que CBC produce salidas con mejor distribución e independencia de bits a corto plazo, pero aún puede tener patrones detectables en secuencias más largas.

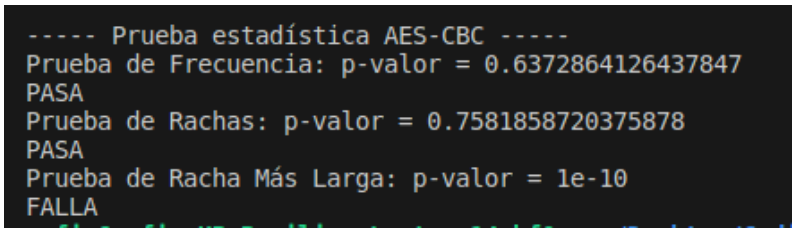

Figura 16: Prueba estadística en modo CBC

- CTR: CTR mostró resultados similares a CBC en las pruebas de Frecuencia y Rachas, pasando ambas con p-valores aceptables. Sin embargo, contrario a lo esperado según la documentación original, también falló en la prueba de Racha Más Larga como se puede ver a continuación.

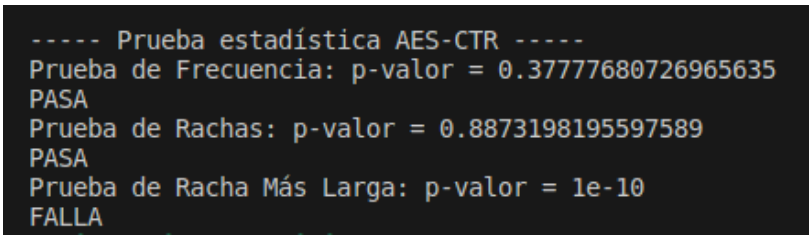

Figura 17: Prueba estadística en modo CTR

Estos resultados muestran patrones interesantes en el comportamiento de los diferentes modos de operación. El éxito en las pruebas de Frecuencia y Rachas indica que los tres modos logran una distribución equilibrada de bits y una buena independencia a corto plazo, cumpliendo con propiedades básicas esperadas de un cifrado seguro. El fallo consistente en la prueba de Racha Más Larga sugiere la presencia de ciertos patrones en secuencias más largas, aunque es importante destacar que la prueba de Rachas normal, que evalúa propiedades más fundamentales de la aleatoriedad, se supera satisfactoriamente en todos los casos. Como señalan Bassham et al. [54], la prueba de Racha Más Larga es particularmente sensible y puede producir fallos incluso en implementaciones criptográficas ampliamente establecidas y seguras. Esta observación es consistente con los hallazgos de Hellekalek [77], quien enfatiza que la interpretación de pruebas estadísticas en criptografía debe considerar el contexto completo del sistema. Esta característica hace que el fallo en esta prueba específica, mientras se mantiene el éxito en las pruebas más fundamentales, no comprometa necesariamente la seguridad global del sistema. Es importante notar que estos resultados estadísticos son solo una parte de la evaluación de seguridad global de un sistema de cifrado, y deben considerarse junto con otros aspectos como la resistencia a ataques conocidos y el cumplimiento de propiedades criptográficas fundamentales.

Estos hallazgos coinciden con las recomendaciones generales en la literatura criptográfica (por ejemplo, Rogaway, 2011 [78]), que generalmente favorecen CBC o CTR sobre ECB para la mayoría de las aplicaciones.

- ECB, aunque rápido, muestra muchas debilidades en términos de seguridad, confirmando su inadecuación para la mayoría de las aplicaciones prácticas.
- CBC ofrece buen equilibrio entre seguridad y rendimiento, siendo especialmente fuerte en términos de difusión.
- CTR muestra un excelente rendimiento combinado con buenas propiedades de seguridad. Su resistencia a ataques en plano y su capacidad de paralelización lo hacen particularmente atractivo para el proyecto.

Para la implementación en FPGA, y usando estos resultados, se consideró que el modo CTR ofrece el mejor equilibrio entre seguridad y rendimiento, especialmente teniendo en cuenta las características específicas del hardware FPGA. La arquitectura de las FPGAs permite un alto grado de paralelización, que el modo CTR puede aprovechar plenamente debido a su independencia entre bloques. Esto permite implementar múltiples instancias del algoritmo AES operando simultáneamente, maximizando el rendimiento. A diferencia de CBC, CTR no requiere implementar tanto el proceso de cifrado como el de descifrado, optimizando así el uso de recursos de la FPGA. CTR también ofrece una latencia constante y predecible para cada bloque, que es muy importante para aplicaciones en tiempo real. En entornos propensos a errores de transmisión, la naturaleza independiente de los bloques en CTR asegura que un error en un bloque no afecte al descifrado de otros. Por último, las FPGAs permiten implementar contramedidas efectivas contra ataques de canal lateral, y la simplicidad de CTR facilita esta tarea sin comprometer significativamente el rendimiento.

7. Curvas elípticas y Diffie-Hellman

En esta sección, se explora el algoritmo de Diffie-Hellman y su implementación con curvas elípticas, una combinación que resuelve el problema de distribución de claves de AES. Esta integración representa un avance significativo en la criptografía moderna, ofreciendo un equilibrio entre seguridad y eficiencia computacional.

7.1. Algoritmo de Diffie-Hellman (DH)

Propuesto en 1976 por los criptógrafos Whitfield Diffie y Martin Hellman en su artículo '*New Directions in Cryptography*' [79], el algoritmo de Diffie-Hellman es uno de los primeros algoritmos de intercambio de claves criptográficas de manera segura a través de un canal de comunicación inseguro. Este trabajo no solo introdujo el concepto de criptografía de clave pública, sino que también sentó las bases para muchos de los sistemas de seguridad que se usan hoy en día.

El principal objetivo de este algoritmo es permitir que dos partes, normalmente llamados Alice y Bob, generen una clave secreta compartida que servirá para cifrar y descifrar mensajes. La seguridad del algoritmo reside en la dificultad de resolver el problema del logaritmo discreto, que plantea un desafío matemático clave. En este contexto, dado un grupo G, y un elemento $y \in$ G se debe encontrar el entero n tal que:

$$y^n = y \cdot y \cdot y \cdot y \cdots \text{ para cierto entero } n.$$

Sin embargo, si se conoce un elemento *x* perteneciente al grupo G, encontrar el número entero n que satisfaga la ecuación $y^n = x$ es una tarea fácil. En este contexto, n se denomina el logaritmo discreto de x en base y, expresado como:

$$n = log_y(x) \Leftrightarrow y^n = x$$

El procesamiento de este algoritmo se puede dividir en cinco pasos o etapas:

1. Elección de los parámetros públicos: Se elige un primo p con un valor bastante grande y una base generadora g, tal que $g < p$.
2. Generación de las claves privada: Ambas partes, Alice y Bob, eligen un número aleatorio a y b dentro del conjunto de enteros menores que p.

3. Cálculo de las claves públicas: Tanto Alice como Bob calculan sus respectivas claves públicas de forma que:

$$clave\ pública\ de\ Alice\ \ A\ =\ g^a\ mod\ p$$
$$clave\ pública\ de\ Bob\ B\ =\ g^b\ mod\ p$$

4. Intercambio de clave: En esta etapa los involucrados intercambian sus claves públicas a través del canal inseguro.

5. Cálculo de la clave secreta compartida: Por último, cada parte calcula el secreto compartido usando la clave pública del otro.

$$clave\ secreta\ de\ Alice\ \ s\ =\ B^a\ mod\ p$$
$$clave\ secreta\ de\ Bob\ \ s\ =\ A^a\ mod\ p$$

   Al final, ambas partes deben tener el mismo valor s, que puede ser utilizado como clave para un cifrado simétrico.

Como se ha mencionado anteriormente, la seguridad de Diffie-Hellman se basa en la dificultad de calcular el logaritmo discreto en un grupo multiplicativo finito. Específicamente, aunque es fácil calcular $g^a\ mod\ p$ y $g^b\ mod\ p$, es computacionalmente difícil determinar a o b dados A y B sin conocer los valores secretos de las claves privadas a y b.

La elección de Diffie-Hellman radica en su simplicidad conceptual y su robustez matemática. Ha resistido décadas de análisis criptográfico y sigue siendo un componente fundamental en muchos protocolos de seguridad modernos.

7.2 Curvas elípticas en la criptografía

Para poder definir qué es exactamente una curva elíptica se debe primero hablar de las curvas algebraicas. Matemáticamente, las curvas algebraicas se definen como: “Dado un polinomio $p\ \in\ K[x, y]$ sobre cierto cuerpo K, una curva algebraica es el conjunto de puntos $(x1, y1)$ que satisfacen que $p(x1, y1)\ =\ 0$ ”

Estas curvas se pueden clasificar por su grado, es decir, el mayor exponente de las variables de la ecuación polinómica que define la curva. En esta sección se hablará únicamente de los 4 primeros grados, siendo el primero el grado 1.

- Grado 1: Curvas lineales

Las curvas de grado 1 son líneas rectas en el plano euclidiano, definidas por una ecuación polinómica que tiene la forma: $ax + b = 0$.
Aunque es un concepto fundamental en la geometría algebraica, tiene una importancia significativa en diversas ramas de las matemáticas aplicadas.

- Grado 2: Curvas cónicas
  Las curvas de grado 2 o curvas cónicas, son un tipo de curva algebraica que están definidas por ecuación polinómicas de grado dos. Su nombre viene dado por ser soluciones generales de ecuaciones cuadráticas en el plano cartesiano. Su ecuación general es de la siguiente forma: $Ax^2 + Bxy + Cy^2 + Ey + F = 0$ siendo A, B, C, D, E y F constantes reales (o complejas, dependiendo del contexto) y "x" e "y" las variables independientes.

  Dentro de las curvas cónicas se pueden hacer una clasificación según los valores de los coeficientes en la ecuación general:
  - Elipses: Se dan cuando en su ecuación se cumple que: $b^2 - 4ac < 0$.
  - Hipérbola: Se produce cuando en su ecuación $b^2 - 4ac > 0$ y $a \neq c$.
  - Parábola: Ocurre cuando $b^2 - 4ac = 0$ y $a \neq 0$.
  - Lineales (por degeneración): Estas se producen cuando alguna de las constantes de la ecuación de una curva cónica puede degenerar en una línea recta, dos líneas que se cruzan, o incluso un punto.

- Grado 3: Curvas cúbicas
  Las curvas cúbicas están definidas por ecuaciones polinómicas de tercer grado de la forma: $f(x) = ax^3 + bx^2 + cx + d$.
  A diferencia de las anteriores, las curvas cúbicas incluyen una serie de propiedades que cabe destacar:
  - Número de Intersecciones: Las curvas de grado 3 pueden tener hasta 3 intersecciones con una línea recta en el plano, ya que son curvas de tercer orden.
  - Puntos Singulares: Al igual que con otras curvas algebraicas, las curvas de grado 3 pueden tener puntos singulares donde las derivadas parciales con respecto a "x" e "y" se anulan simultáneamente.
  - Clasificación por Tipo de Singularidades: Las curvas de grado 3 se clasifican según el tipo de singularidades que pueden tener, como puntos de cruce, puntos de tangencia o puntos de inflexión.

- Grado 4 o superior

  Este tipo de curva curvas están definidas por ecuaciones polinómicas de cuarto grado o superior, cuya forma es: $ax^4 + bx^3 + cx^2 + dx + e = 0$.

  Estas curvas añaden un mayor número de intersecciones a sus propiedades, así como puntos singulares, además de añadir una topología y un comportamiento geométrico más complejo.

Una vez vistas las curvas algebraicas, se puede definir una curva elíptica como una curva algebraica plana de grado 3 o cúbica sobre un cuerpo K. En la figura inferior, se observa una representación gráfica de una curva elíptica, la cual refleja la forma de este tipo de curvas.

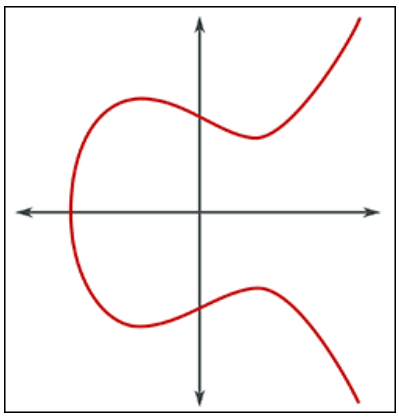

Figura 18: Representación de las curvas elípticas (Source: Efficient ECC-Based Authentication Scheme for Fog-Based IoT Environment)

Además, toda curva elíptica admite su expresión en forma de ecuación de Weierstrass: $y^2 = x^3 + Ax + B$, donde A y B son coeficientes que definen la curva elíptica específica. La ecuación de Weierstrass será el polinomio que genera la curva elíptica.

La ecuación de Weierstrass introduce una característica importante, la invariante discriminante, definida por: $\Delta = -16(4a^3 + 27b^2)$. Este discriminante determina las propiedades de la curva elíptica, incluyendo la presencia de puntos singulares o puntos cuspidales. Dependiendo del valor de Δ, se puede obtener:

- Si $\Delta \neq 0$, la curva elíptica es no singular, lo que significa que no tiene puntos singulares. En este caso, la curva es suave y tiene una estructura de grupo bien definida.
- Si $\Delta = 0$, la curva elíptica es singular, lo que significa que tiene al menos un punto singular. Los tipos de singularidades pueden incluir:
  - Nodos: donde dos ramas de la curva se cruzan.
  - Cúspides: donde la curva tiene un punto agudo.

Estas propiedades son fundamentales para el estudio de curvas elípticas en teoría de números y criptografía, ya que la estructura del grupo de puntos en una curva no singular es esencial para aplicaciones como el intercambio de claves Diffie-Hellman y la firma digital [80].

En definitiva, el uso de las curvas elípticas en el campo de la criptografía permite beneficiarse de las propiedades de éstas para ofrecer sistemas de encriptado con el mismo nivel de seguridad de otros sistemas como RSA pero con mayor eficiencia [81]. Las curvas elípticas destacan especialmente en las siguientes áreas:

- Eficiencia en el tamaño de la clave: Las claves públicas tienen una longitud considerablemente más corta que las claves equivalentes en otros sistemas como RSA. Por ejemplo, una clave ECC de 256 bits proporciona un nivel de seguridad comparable a una clave RSA de aproximadamente 3072 bits.
- Rendimiento y velocidad: Debido a la naturaleza algebraica de las curvas elípticas, las operaciones criptográficas como la multiplicación de puntos (explicada en el siguiente punto) pueden realizarse de manera más rápida en comparación con otros métodos. Esto no solo mejora la velocidad de las comunicaciones seguras, sino que también reduce el consumo de energía.
- Seguridad. Ofrecen un nivel de seguridad robusto debido a la dificultad computacional en resolver problemas como el logaritmo discreto. Además, la estructura matemática de las curvas elípticas también proporciona resistencia frente a ciertos tipos de ataques criptoanalíticos.

Este estudio se centrará en la aplicación de ECC para la generación de claves públicas y privadas junto con el intercambio de claves.

7.3 Diffie-Hellman con curvas elípticas (ECDH)

Como se ha mencionado anteriormente, la criptografía asimétrica consiste en la generación de un par de claves: la clave pública y la clave privada. Se usarán este par de claves para realizar el cifrado y descifrado de la información.

En ECC, se utilizan diversas curvas que han sido ampliamente estudiadas y son reconocidas por su aplicación en entornos de ciberseguridad. Algunas de estas son: curve 448 [82], curve secp256r1 (también conocida como curve p-256 o prime256v1) [83], Curve25519 [84],

ED25519 [85], Elliptic Curve Fully Hashed Menezes-Qu-Vanstone [86], secp192k1 [87] entre otras.

Para este estudio se hablará específicamente de las curvas Curve25519 y secp256r1 debido a sus características destacadas en seguridad y eficiencia computacional. La primera es conocida por su diseño optimizado que garantiza un alto nivel de seguridad y rendimiento en aplicaciones como el intercambio de claves. Por otro lado, secp256r1, es ampliamente adoptada en la seguridad TLS/SSL y ofrece un buen equilibrio entre seguridad y eficiencia. Comparadas con otras curvas, se eligen estas dos por su resistencia a ataques criptoanáliticos y su capacidad para operar eficientemente en dispositivos con recursos limitados.

La elección de ECDH en combinación con AES-128 proporciona un equilibrio óptimo entre seguridad y eficiencia computacional. Una clave ECDH de 256 bits ofrece una seguridad comparable a una clave RSA de 3072 bits, mientras que AES-128 proporciona un nivel de protección robusto con menor consumo de recursos que AES-256. Esta configuración mantiene una alta resistencia contra ataques conocidos, incluidos los ataques cuánticos teóricos actuales, al tiempo que optimiza el rendimiento del sistema.

La Figura 19 ilustra el proceso de intercambio de claves Diffie-Hellman en el contexto de ECC, mostrando cómo dos partes pueden establecer una clave compartida de manera segura utilizando sus claves privadas y públicas.

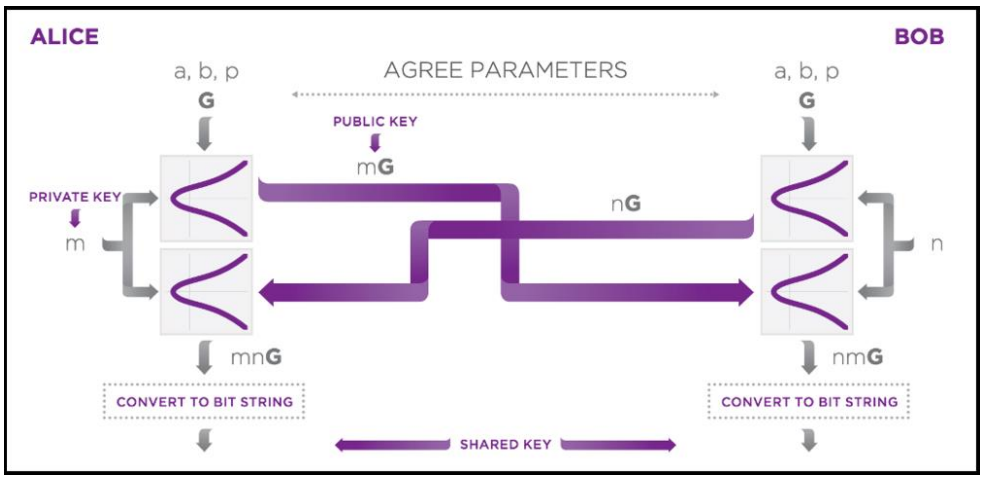

Figura 19: Diagrama de intercambio de claves Diffie-Hellman (Source: The Elliptic Curve Diffie Hellman KE Protocol & Its Mathematical DLP Analysis)

A continuación, se analizan con más detalle las dos curvas mencionadas.

7.3.1 Parámetros de las curvas

Las curvas, como se ha mencionado anteriormente tienen varios parámetros definidos [88]. Estos son:

- P: define el campo finito de la curva. Es un conjunto finito de elementos el cual permite realizar las operaciones aritméticas necesarias de forma que los resultados obtenidos de dichas operaciones siempre pertenezcan al conjunto establecido por el campo finito. Un campo finito F_P tiene una cantidad de q elementos, siendo P una potencia de un número primo, es decir, $P = q^n$, con q siendo otro número primo y n un entero positivo.
  El tamaño del cuerpo tiene un gran impacto en el nivel de seguridad, puesto que cuanto más grande es el cuerpo definido mayor es la dificultad de resolver los algoritmos, como el problema del logaritmo discreto.

- "A" y "B": Son parámetros fundamentales de la curva que definen la forma de esta. Como se ha mencionado en la sección anterior, una curva elíptica puede ser expresada mediante la ecuación de Weierstrass, que se recuerda, es de la forma: $y^2 = x^3 + Ax + B$, donde A y B son elementos del campo finito F_P. Sin embargo, estos dos parámetros deben satisfacer una ecuación que garantice que la curva no tenga puntos singulares (invariante discriminante), esto es: $4a^3 + 27b^2 \neq 0$.

  Al final, los puntos (x, y) que satisfacen la ecuación de la curva, junto con un punto en el infinito $O(0,0)$, formarán el grupo abeliano que, con A y B establece la estructura del grupo.

- "G" o generador: Es un punto perteneciente a la curva elíptica E en el campo finito F_P empleado para generar o producir un grupo cíclico de puntos de la curva. Dado que la selección de un generador adecuado es de gran importancia respecto a la seguridad y eficiencia de los algoritmos, hay ciertas consideraciones que se deben tener en cuenta:

  - Para un generador G de una curva elíptica, el Orden del punto generador es un número entero n tal que $nG = O$, siendo O el punto en el infinito. Para su uso criptográfico, el orden del generador debe ser un primo grande, dificultando con ello la resolución del problema del logaritmo discreto.

- El generador G debe ser capaz de generar un grupo cíclico que incluya la mayor cantidad de puntos de la curva elíptica, aprovechando así, el máximo de las propiedades de la curva.
- En la práctica, lo más usado es la aplicación de puntos generadores estandarizados dado que estos generadores ya han sido analizados y estudiados por la comunidad criptográfica y organizaciones como el NIST.

.

- Orden del punto generador: Como su nombre indica, el orden de un punto G en una curva elíptica E sobre el campo finito F_P es el número más pequeño tal que: $nG = O$. Este parámetro es crucial en criptografía, ya que dificulta el cálculo del problema del logaritmo discreto. En la generación de claves criptográficas, la clave privada es un número aleatorio en el rango [1, n-1], asegurando así la seguridad del sistema.
  Usar un valor de n lo suficientemente grande permite garantizar que la curva pueda generar una amplia gama de posibles claves privadas, permitiendo que dos usuarios puedan tener la misma clave privada.

- Cofactor: Se define como la relación entre el número de puntos de la curva r y el orden del punto generador: $h = \frac{r}{n}$.
  Aunque no es un valor directamente empleado en los cálculos de la curva, el valor obtenido en el cofactor permitirá conocer cuántos subgrupos de orden n existen en el grupo completo de puntos de la curva.
  Dado que un valor alto para implica introducir vulnerabilidades, Idealmente se busca que $h = 1$, que minimiza la existencia de subgrupos pequeños que podrían ser explotados en ataques criptográficos.
  Adicionalmente, cobra más importancia cuando se pretende implementar algoritmos criptográficos basados en curvas elípticas sobre hardware. Esto se debe a que un cofactor pequeño puede llegar a simplificar ciertos cálculos.

7.3.2 Obtención de claves

El primer paso en la criptografía asimétrica es la generación de las claves. Haciendo uso de ECC, la generación de la clave privada K del usuario A (Alice) se basa en obtener un valor que cumpla la condición: $1 \leq d \leq n - 1$, donde se ha visto que n es el punto del generador.

Sin embargo, la obtención de la clave pública de A requiere de más complejidad y resolución de problemas criptográficos. Para ello se deben introducir los conceptos de operaciones con puntos, de las cuales se destacarán la suma de dos puntos y la multiplicación de un punto por un escalar, dado que se han empleado en la implementación de las mencionadas curvas 256r1 y Curve25519.

7.4. Matemáticas de las curvas elípticas

En este apartado, se explicarán los fundamentos matemáticos que sustentan las curvas elípticas.

7.4.1 Suma de puntos

La operación principal empleada en el cálculo de puntos sobre las curvas elípticas es la suma de puntos, una operación que tiene ciertas particularidades. La operación de suma de dos puntos pertenecientes a una curva elíptica permite formar la base del grupo abeliano.

Aunque solo se ha mencionado, los grupos abelianos hacen referencia al conjunto de puntos en una curva elíptica $E(F_p)$ que cumplen ciertas propiedades algebraicas como son:

- El cierre: Para todo $a, b \in G, a * b \in G$
- Asociación: Para todo $a, b, c \in G, (a * b) * c = a * (b * c)$
- Elemento Identidad: Existe un elemento $e \in G$ tal que para todo $a \in G, a * e = e * a = a$
- Elemento inverso: Para $a \in G$, existe un elemento $b \in G$ tal que $a * b = b * a = e$
- Conmutatividad: Para todo $a, b \in G, a * b = b * a$

La suma de dos puntos de la curva también es el primer paso para la multiplicación de un escalar K por un punto de la curva Q, $K * Q$, ya que se define como la suma del punto Q consigo mismo un número de veces K.

Este algoritmo tiene una serie de consideraciones a tener en cuenta, como se ve en la Figura 20. Considerando que se tienen los puntos de la curva P y Q, y el punto neutro (infinito) O:

- Consideración 1: $P = O$ ó $Q = O$

Cuando la suma de dos puntos implica que uno de los puntos esté en el infinito, se debe aplicar las propiedades especiales de la estructura del grupo elíptico. Es decir, el punto del infinito actuará de forma aritmética como el punto (0, 0) y por consiguiente el resultado de la operación será el punto que no sea neutro. Por ejemplo:

$$P = 0 \quad \rightarrow \quad P + Q = O + Q = Q$$

- Consideración 2: $P = Q$

  Cuando los dos puntos de la curva que se intenta sumar resultan iguales, lo que se hace es el doble del punto. Esta operación requiere que sea calculable la pendiente de la tangente a la cuerva para, a continuación, calcular las coordenadas del nuevo punto:

$$Pendiente\ m = \frac{3{x_1}^2 + a}{2y_1}$$ *(a es definido por el parámetro 'a' de la curva)*

$$Nueva\ coordenada\ x_3 = m^2 - x_1 - x_2$$
$$Nueva\ coordenada\ y_3 = m(x_1 - x_3) - y_1$$

  Es importante que como se está trabajando en el contexto de curvas elípticas, se debe realizar el módulo sobre de los resultados sobre el cuerpo definido por la curva. Es decir:

$$m = \frac{3{x_1}^2 + a}{2y_1} \ mod\ p$$
$$x_3 = m^2 - x_1 - x_2 \ mod\ p$$
$$y_3 = m(x_1 - x_3) - y_1 \ mod\ p$$

- Consideración 3: $P \neq Q$

  En el caso de que los puntos que se van a sumar son distintos entre sí, se aplica que el punto resultante sea R = P + Q. Para ello se debe calcular la pendiente de una forma distinta al caso 2.

$$m = \frac{y_2 - y_1}{x_2 - x_1} \ mod\ p$$

  La obtención de x_3 e y_3 resulta ser la misma que el caso 2.

Como se observa en la Figura 21, la representación gráfica de la suma de dos puntos en una curva elíptica ayuda a visualizar estas consideraciones. Esta imagen ilustra cómo la suma de dos puntos de la curva genera un tercer punto, reforzando así el concepto de grupo abeliano en este contexto.

```
AddPoints Funtion

Input: puntos P y Q
Output: Punto R

begin
    if P = O then
        Return Q
    elsif Q = O then
        return P
    elsif P = Q then:
        Pendiente m = 3x12 + a2y1 mod p
    else :
        Pendiente m = y2 - y1x2 -x1 mod p

    Rx =  m2 - x1- x2   mod p
    Ry = m(x1 - x3) - y1   mod p

    Return Rx & Ry
end
```

Figura 20: *pseudocódigo algoritmo suma de dos puntos de la curva*

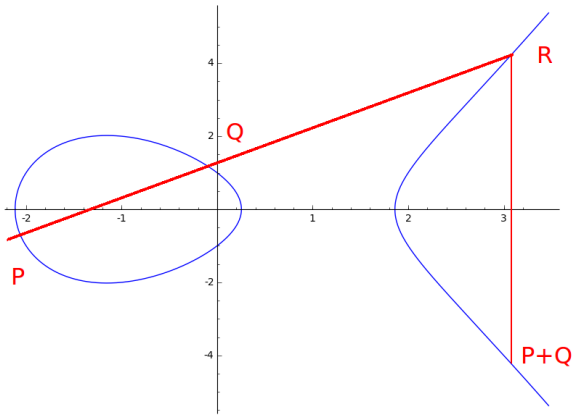

Figura 21: *Suma de dos puntos de la curva elíptica (Source: Secure Computation Protocol of Text Similarity against Malicious Attacks for Text Classification in Deep-Learning Technology)*

7.4.2 Multiplicación de un punto por un escalar

La operación de multiplicar un punto de la curva P por un escalar K es equivalente a sumar el punto P consigo mismo un número de veces K. Y como se ha visto antes, esto recuerda a la operación de doblado de puntos (suma de puntos, caso 2), aunque se deben tener unas ciertas consideraciones a tener en cuenta.

La implementación de este algoritmo, ilustrada en la Figura 22, requiere que el escalar K se represente de forma binaria, y como se usa la suma de dos puntos, se inicializa el punto R = O y el punto P = Q. Una vez se han inicializado los datos se procede a

recorrer el array que almacena el escalar en forma binaria y para cada iteración se realiza el doblado del punto Q, es decir R = 2Q.

En caso de que el escalar en la posición 0 ≦ i ≦ (scalar'length - 1) tenga su bit a '1', se procederá a sumar los puntos R y Q, de forma que R = Q + R.

```
Multiply Point by scalar Function

Input: Point P, Scalar K
Output: Point R

begin
    Initialize R = O
    Initialize binary_k = binary(K)

    for i in 0 to binary_k.lenght-1 loop
        R  = AddPoints (P, P)       //doble del punto
        if binary_k(i) = 1 then
            R = AddPoints (R, P)    //suma de puntos
        end
    end loop
    return R
end
```

Figura 22: *pseudocódigo algoritmo de multiplicación de un punto por un escalar*

Una vez definidas las funciones y operaciones necesarias para la criptografía asimétrica usando ECC, se puede obtener la clave pública de A usando el punto generador de la curva G y la clave privada de A.

La forma de comprobar que la clave pública obtenida pertenece a la curva es intentar resolver la ecuación de tercer grado que la define.

En este caso, se tienen que las ecuaciones son:

para la curva secp256r1: $y^2 = x^3 + ax + b \bmod p$

para la Curve25519: $y^2 = x^3 - ax^2 + x \bmod p$

Aunque, se debe tener en cuenta que por el propio diseño de la Curve25519, Daniel J. Bernstein, autor de *Curve25519: new Diffie-Hellman speed records* [89] indica que "los tiempos de Curve25519 incluyen compresión de claves gratuita, validación de claves gratuita y protección avanzada contra ataques de temporización". En este contexto, 'gratuita' se refiere a que estas operaciones no requieren tiempo adicional de cómputo. Por ende, no hace falta que se verifique que el punto obtenido pertenezca a la curva,

pero si se hace especial atención en la sección 3 del artículo, "*Responsibilities of the user*" [89, p.3] que la seguridad recae en la responsabilidad del usuario y como se debe modificar la clave privada d $(1 \leq d \leq n-1)$.

7.4.3 Operaciones de Montgomery

La Curve25519 utiliza una serie de operaciones específicas para realizar cálculos eficientes dentro del protocolo Diffie-Hellman, las cuales son llamadas operaciones de Montgomery. Estas operaciones se centran en la aritmética modular sobre cuerpos finitos, optimizadas para trabajar con números primos de 256 bits [90]. Entre las más importantes se encuentran:

- Suma de Puntos (*Montgomery Ladder*): En esta curva, la suma de puntos se realiza utilizando el algoritmo *Montgomery Ladder*, que es una técnica eficiente para realizar multiplicaciones escalares. En lugar de la tradicional suma y duplicación, utilizada en otras curvas elípticas, *Montgomery Ladder* utiliza operaciones especiales basadas en la aritmética modular para mejorar la eficiencia computacional.

- Doble de Puntos: El doble de puntos en la Curve25519 también se lleva a cabo mediante métodos específicos de Montgomery, que aprovechan las propiedades algebraicas particulares de las curvas elípticas de Montgomery. Estos métodos permiten realizar la operación de manera más rápida y eficiente en comparación con otros enfoques.

Para una descripción detallada de las operaciones, se puede consultar el Apéndice B.

7.5 Problemas y desventajas de utilizar ECDH

Al implementar ECDH, es importante considerar las siguientes limitaciones y desafíos:

1. Complejidad de implementación: ECDH requiere una implementación cuidadosa para evitar vulnerabilidades. Errores en la implementación pueden llevar a debilidades en la seguridad del sistema.

2. Selección de curvas: La elección de la curva adecuada es crucial y puede ser compleja. Diferentes curvas ofrecen distintos niveles de seguridad y eficiencia, y la

elección debe basarse en las necesidades específicas de la aplicación, en este caso, se requiere de claves que sean fáciles de implementar en hardware.

3. Generación de claves: Es fundamental generar claves verdaderamente aleatorias y seguras [91]. Una generación de claves débil puede comprometer todo el sistema.

4. Validación de puntos: Es necesario validar que los puntos estén en la curva para evitar ataques de curva inválida. Sin embargo, esta validación puede añadir complejidad y tiempo de procesamiento.

5. Coste computacional: Aunque ECDH es más eficiente que RSA para el mismo nivel de seguridad, aún puede ser computacionalmente costoso para algunos dispositivos de recursos limitados.

6. Estandarización: Aunque existen estándares para ECDH, la variedad de curvas y parámetros puede llevar a problemas de interoperabilidad entre diferentes implementaciones.

7. Resistencia cuántica: Aunque ECDH es más resistente a los ataques cuánticos que RSA, aún no se considera completamente seguro contra computadoras cuánticas potentes.

Abordando inicialmente la Curve25519, el primer problema que se encuentra es referido al usuario. Daniel J. Bernstein indica, como se ha mencionado anteriormente, que el usuario final y legítimo es el responsable de generar una clave secreta uniforme y aleatoria de 32 bytes dentro del rango n de la curva. No cumplir esta condición eliminaría totalmente la seguridad de la curva. También se asume que el usuario mantendrá su clave privada de forma confidencial y no se usará de forma inapropiada, es decir, el uso de la clave privada únicamente se limitará a la generación de la clave pública y el cálculo del secreto compartido entre “A” y “B”. Por último, el usuario no debería desechar la clave privada tras un único uso, la clave privada d, puede usarse con diferentes claves públicas. Esto tiene un impacto en el rendimiento puesto que generar un nuevo par de claves para cada interacción puede ser costoso e innecesario computacionalmente en la mayoría de casos, y reusar la misma clave con diferentes claves de otros usuarios producirá un secreto compartido único entre ambos usuarios.

La curva elíptica 25519 puede ser segura, pero debe ir acompañada de un criptosistema de clave secreta fuerte, en el caso de este estudio Diffie-Hellman. Suponemos que se tiene un usuario con intenciones moralmente cuestionables, el atacante. El atacante puede visualizar las claves públicas de “A” y “B” (q_1 y q_2) junto con los mensajes cifrados por el criptosistema.

Dicho atacante, con la intención de descifrar o falsificar mensajes puede generar su propia clave pública (q_0) de manera no estándar sin ser verificada por los usuarios legítimos. Es decir, el atacante habiendo interceptado la comunicación entre A y B, ha falsificado la información usando claves públicas no legítimas usando el criptosistema C y la Curve25519 (ataque *man in the middle*).
Es por esto que la seguridad del sistema depende también del criptosistema elegido. Ser consciente de los posibles vectores de ataque, como la generación de claves públicas no estándar, ayuda a diseñar sistemas más seguros. Validar claves públicas y asegurar la implementación adecuada de funciones de derivación de claves y criptosistemas son pasos esenciales para mantener la seguridad.

Por otro lado, la curva secp256r1, aunque no tiene vulnerabilidades prácticas conocidas, se ha criticado mucho por su diseño. La curva fue estandarizada por la NSA (*National Security Agenc*y, Agencia de Seguridad Nacional) [92] de los Estados Unidos, pero no aportaron los detalles de su decisión como se puede encontrar en el caso de Daniel J. Bernstein y la Curve25519. Esto ha suscitado especial preocupación con los años a la comunidad de criptógrafos debido a que en 2013 se descubrió que la NSA había implementado un backdoor (puerta trasera) en el Dual_EC_DRBG [93] (*Dual Elliptic Curve Deterministic Random Bit Generator*), un algoritmo que permite generar números criptográficamente pseudoaleatorios seguros usando metodos en criptografía de curvas elípticas. Este descubrimiento sobre el Dual_EC_DRBG permitía a la NSA predecir los números aleatorios generados por el sistema, comprometiendo así la seguridad de sistemas y aplicaciones que lo implementan. Por esta misma razón se ha llegado a especular que la curva pueda tener una puerta trasera.

A pesar de estas limitaciones, ECDH sigue siendo una opción robusta y eficiente para el intercambio de claves cuando se implementa correctamente y se elige la curva adecuada.

7.6 Implementación y pruebas en Python

En este punto se detallarán nuestra implementación ECDH con las curvas seleccionadas, Curve25519 y secp256r1 en Python, así como las pruebas realizadas para verificar su correcto funcionamiento.

7.6.1 Detalles de implementación

La implementación se centra en dos curvas elípticas mencionadas. Cada una de estas tiene sus propias particularidades de implementación.

La Curve25519 fue diseñada para ECDH. Además, utiliza coordenadas de Montgomery y no requiere de validación. Por otro lado, Secp256r1 se implementa usando coordenadas afines estándar y requiere de validación de puntos.

Para ambas curvas se implementan las operaciones fundamentales: suma de puntos, duplicación de puntos y multiplicación escalar.

Aspectos clave de la implementación:

- Aritmética modular: Todas las operaciones se llevan a cabo en el campo finito definido por un número primo asociado a la curva elíptica.
- Representación de puntos: Se utilizan tuplas (x, y) para representar puntos en la curva.
- Manejo del punto en el infinito: Se representa este punto con (0, 0) en las operaciones.
- La Curve25519, a diferencia del AES y la curva Secp256r1, funciona en *Little-Endian*, lo cual precisó de un tratamiento específico.

7.6.2 Función de derivación de claves (KDF)

Se implementa una función de derivación de claves basada en HKDF [94] (*HMAC-based Key Derivation Function*) para convertir el secreto compartido ECDH en una clave adecuada para AES-128. Este paso es crucial por varias razones:

- Uniformidad: Asegura que la clave tiene una distribución uniforme.
- Longitud adecuada: Adapta la longitud del secreto compartido a la longitud requerida para AES-128.
- Seguridad adicional: Añade una capa extra de seguridad, haciendo más difícil recuperar el secreto original incluso si se compromete la clave AES.

Algunos detalles clave de la implementación son:

- Se utiliza SH-256 como función *hash* subyacente para HKDF.
- La longitud de la clave derivada se establece en 16 bytes (128 bits), que es exactamente lo que AES requiere.
- No se utiliza un valor de *salt* (un dato aleatorio que se añade para aumentar la seguridad llamado sal en inglés) en esta implementación; sin embargo, en un entorno de producción, es recomendable usar un valor de sal único para cada derivación de clave.

7.6.3 Desafíos y errores

Aunque los principales problemas que se encuentran al principio están derivados de un mal entendimiento de la criptografía con curvas elípticas, no se tuvo mucha dificultad a la hora de realizar la implementación en Python de Secp256r1 dado que resultó sencillo realizar las operaciones criptográficas de suma de puntos y multiplicación por escalar. Sin embargo, sí existieron ciertos inconvenientes en la implementación de la Curve25519 tanto en Python como en VHDL.

Específicamente en este estudio, se ha realizado una función de verificación de claves la cual consiste en satisfacer la ecuación de la curva usando la clave pública recibida. Esta verificación permite verificar si las claves públicas recibidas son legítimas, pero no garantiza una protección completa, ya que un atacante podría seguir interponiéndose en la comunicación si genera sus claves utilizando la curva correctamente. Solo se puede detectar y descartar el ataque cuando las claves no son legítimas (es decir, cuando no satisfacen la ecuación de la curva).

7.6.4 Pruebas y verificación

Para garantizar la robustez y seguridad de la implementación de ECDH con las curvas Curve25519 y secp256r1, se han desarrollado y ejecutado un conjunto exhaustivo de pruebas. Estas pruebas no solo validan la correcta implementación de los algoritmos, sino que también verifican su comportamiento en diversos escenarios, incluyendo casos límite y potenciales situaciones de error.

- Pruebas de integración: Se verifica el proceso completo de ECDH, desde la generación de claves hasta el cálculo del secreto compartido. Esto incluye: generación de claves privadas y públicas, intercambio de claves y cálculo y verificación del secreto compartido, como se observa en la Figura 23, que muestra los resultados de las pruebas de integración para las curvas P-25, y la Figura 24 para la Curve25519.

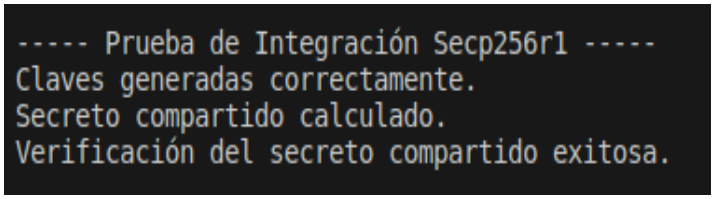

Figura 23: *Prueba de integración de la curva p256*

```
----- Prueba de Integración Curve25519 -----
Secreto compartido de Alice: 46898f37cb7780fe574ed156850d97f8ffa3fa7bd6273e285785537e9f848c45
Secreto compartido de Bob: 46898f37cb7780fe574ed156850d97f8ffa3fa7bd6273e285785537e9f848c45
Verificación del secreto compartido exitosa.
```

Figura 24: *Prueba de integración de la Curve25519*

- Verificaciones de seguridad: Que incluye una comprobación de que las claves privadas están dentro del rango válido. Para Curve25519, se verificó adicionalmente que las longitudes de clave son correctas (32 bytes) y que la función de ajuste funciona correctamente. Para secp256r1, se realizó la verificación de que las claves públicas no son el punto en el infinito y la validación de que los puntos (claves públicas y secretos compartidos) están en la curva, como se muestra en la Figura 25 y 26, que ilustra los resultados de las verificaciones de seguridad realizadas.

```
----- Verificaciones de Seguridad Secp256r1 -----

Claves privadas en rango válido.
Claves públicas no son el punto en el infinito.
Claves públicas están en la curva.
Secretos compartidos están en la curva.
```

Figura 25: *Prueba de seguridad para la curva p256*

```
----- Verificaciones de Seguridad Curve25519 -----
Longitudes de clave correctas.
La clave privada está en el rango válido.
La clave pública está en el rango válido.
La función de ajuste funciona correctamente.
```

Figura 26: *Prueba de seguridad para la curva 25519*

- Pruebas con vectores de prueba NIST: Se utilizan vectores de prueba proporcionados por el NIST, al igual que en la implementación de AES, para verificar la correcta implementación de las curvas. Estos vectores incluyen entradas y salidas conocidas para operaciones clave, permitiendo validar la implementación con un estándar reconocido.

- Pruebas de casos límite: Se implementan pruebas para manejar casos especiales. Para secp256r1, se realizaron pruebas de suma con el punto en el infinito, multiplicación por 0 y por el orden de la curva, como se muestra en las Figura 27 y 28, que presentan los resultados de las pruebas de casos límite realizadas.

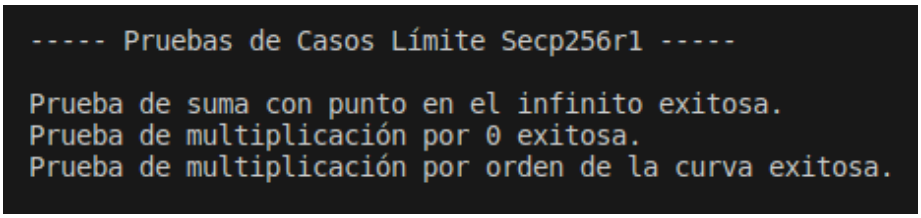

Figura 27: *Prueba de caso límite sobre la curva p256*

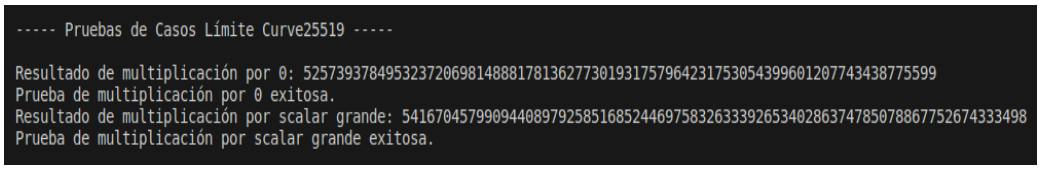

Figura 28: *Prueba de caso límite sobre la Curve25519*

- Pruebas de rendimiento: Se ejecutan pruebas comparativas entre Curve25519 y secp256r1 para evaluar y comparar su eficiencia en términos de tiempo de ejecución y uso de recursos. Los resultados, que se presentan en las Figuras 29 y 30, muestran que Curve25519 es significativamente más rápida, capaz de generar aproximadamente 492 pares de claves por segundo y calcular 496 secretos compartidos por segundo. En comparación, Curvep256 puede generar alrededor de 81 pares de claves por segundo y calcular 79 secretos compartidos por segundo.

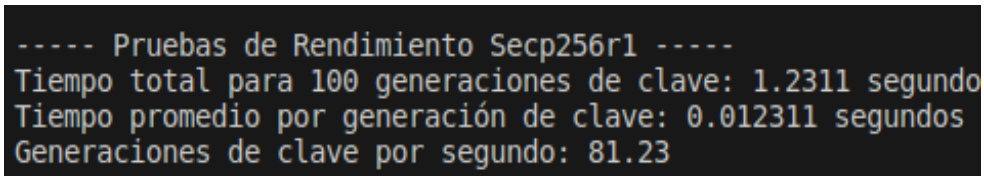

Figura 29: *Prueba de rendimiento de la Curvep256*

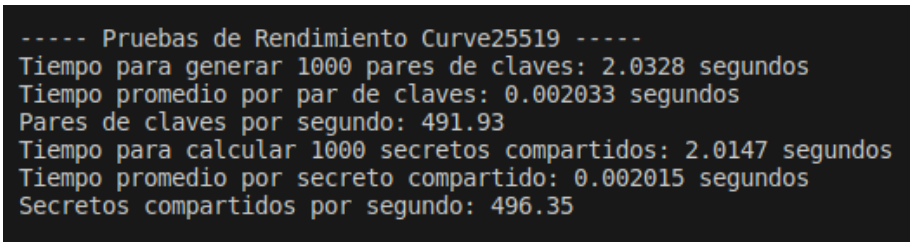

Figura 30: *Prueba de rendimiento de la Curve25519*

- Pruebas de manejo de errores: Ambas implementaciones demostraron un manejo adecuado de errores para entradas inválidas, como claves de longitud incorrecta o fuera de rango.

- Pruebas de resistencia a ataques de *timing*: Los ataques de *timing* son técnicas que explotan las variaciones en el tiempo de ejecución de las

operaciones para intentar extraer información sensible. Ambas implementaciones mostraron una diferencia de tiempo promedio muy pequeña entre operaciones, lo que sugiere una buena resistencia a este tipo de ataques. Curve25519 mostró una diferencia promedio de 0.000361 segundos, mientras que secp256r1 mostró una diferencia de 0.000068 segundos.

- Pruebas de la función KDF: Se verifica que la función de derivación de claves (KDF) produce salidas del tamaño correcto (128 bits para AES-128) y con la distribución estadística esperada.

Todas las pruebas se ejecutaron con éxito, confirmando la correcta implementación de ECDH para ambas curvas que estaban ya listas para ser implementadas en VHDL. Los vectores de prueba del NIST produjeron los resultados esperados, validando la conformidad de la implementación con los estándares establecidos.

En cuanto a las pruebas de rendimiento, se mostró que Curve25519 ofrece un mejor rendimiento en comparación con secp256r1, especialmente en la generación de claves y el cálculo del secreto compartido, lo cual es consistente con las características de diseño de esta curva. Las pruebas de resistencia demostraron que la implementación maneja adecuadamente las entradas inválidas.

Esta implementación y pruebas ha proporcionado una comprensión profunda de los principios subyacentes de ECDH y las curvas elípticas, sentando una base sólida para su implementación en VHDL.

8. Implementación criptográfica en FPGA

La implementación de algoritmos criptográficos en hardware, específicamente en FPGAs, ofrece ventajas significativas en términos de rendimiento y seguridad. Este capítulo detalla la implementación, abordando la elección del hardware, el diseño de la arquitectura, las optimizaciones realizadas y los procesos de validación y pruebas

8.1. Implementación en el Hardware

8.1.1 Descripción del hardware

Gracias al Departamento de Sistemas Informáticos de la Escuela, se presentó la posibilidad de realizar la implementación de los algoritmos en tres FPGAs distintas: Basys 3 [95], Zybo zynq-7000 [96] y Nexys A7 [97]. Cada una de estas placas ofrecen características únicas que las hacen adecuadas para diferentes tipos de proyectos.

- Basys 3: La Basys 3 es una placa con fines educativos y realización de pequeños proyectos debido a su simpleza y facilidad de uso, pero sus limitados recursos podrían ser un obstáculo debido a su almacenamiento y capacidad de procesamiento. En la figura inferior, se muestra la FPGA AMD-Xilinx Basys 3, que cuenta con las siguientes características:
    - Chip lógico Xilinx Artix-7 XC7A35T.
    - Lógica programable con 33280 celdas y 5200 bloques.
    - 1800 kbits de RAM.
    - Conectividad UART, USB, PMOD.
    - Reloj con una frecuencia de 100 MHz.

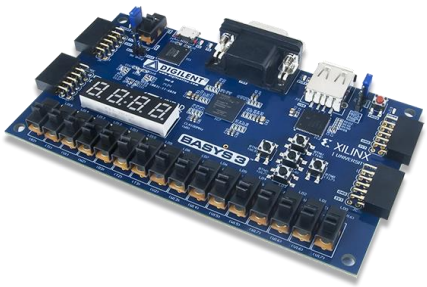

Figura 31: *FPGA AMD-Xilinx Basys 3*

Aunque la propia caja de esta placa menciona que puede llegar a los 450 MHz, esto sucede en zonas localizadas y en casos específicos. Es importante entender que la frecuencia máxima alcanzable depende de múltiples factores, incluyendo la complejidad del diseño implementado y las rutas críticas dentro del circuito.

- Zybo Zynq-7000: Esta placa es una excelente opción para proyectos que requieren la integración de procesamiento software y hardware, como sistemas embebidos y aplicaciones de alto nivel de procesamiento de datos. La presencia del procesador ARM permite la ejecución de sistemas operativos como Linux, lo que abre la puerta a aplicaciones más complejas que combinan software de alto nivel con aceleración hardware. En la Figura 32 se ilustra esta FPGA, la cual incorpora las siguientes características:
    - Chip lógico Xilinx Zynq-7000 XC7Z010.
    - Procesador ARM Cortex-A9 dual core.
    - Lógica programable con 28000 celdas lógicas.
    - 512 MB DDR3L RAM, 128Mb Serial Flash.
    - Conectividad USB, Ethernet, UART, Pmod y HDMI.
    - Frecuencia de reloj de 125 MHz.

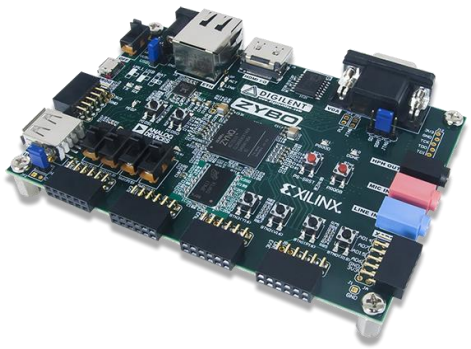

Figura 32: *FPGA AMD-Xilinx Zybo Zynq-7000*

- Nexys A7: Ofrece un balance entre recursos, conectividad y capacidad de procesamiento, lo que la convierte en una opción ideal para una gran variedad de proyectos. La Figura 33 muestra una imagen de esta FPGA, que incluye las siguientes especificaciones principales:
    - Chip Xilinx Artix-7 variante XC7A50T.
    - Lógica programable con 101440 celdas lógicas.
    - 4,860 Kbits de memoria RAM.
    - Conectividad USB, Ethernet, UART, Pmod y HDMI.

- Frecuencia de reloj de 100 MHz.

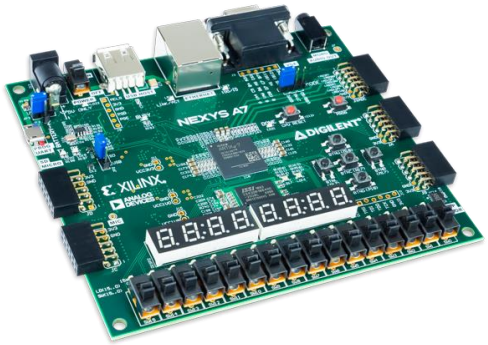

Figura 33: *FPGA AMD-Xilinx Nexys A7*

Para el proyecto, se decidió utilizar la Nexys A7 debido a la cantidad de recursos que dispone. Aunque la conectividad UART es suficiente para el caso de uso, el gran tamaño de memoria RAM permite evitar problemas de cuello de botella al realizar operaciones complejas. A diferencia de la Zybo, la Nexys A7 se enfoca en el procesamiento hardware, lo que la convierte en la mejor opción para la aceleración de algoritmos criptográficos.

8.1.2 FPGA vs CPU

La elección entre FPGA y CPU para la implementación de algoritmos criptográficos depende de varios factores. Las FPGAs ofrecen ventajas significativas en ciertos escenarios como, por ejemplo:

- Diseño de circuitos específicos: Las FPGAs permiten diseñar circuitos dedicados para algoritmos específicos, lo que resulta en una ejecución más eficiente que en una CPU de propósito general.
- Paralelismo: Las FPGAs pueden ejecutar múltiples operaciones simultáneamente, aprovechando su estructura de matrices de bloques lógicos programables. Esto contrasta con las CPUs, que tienen un número limitado de núcleos y una arquitectura fija.
- Eficiencia energética: En aplicaciones que requieren un bajo consumo de energía, como sistemas embebidos o dispositivos IoT, las FPGAs pueden ofrecer un mejor rendimiento por vatio que las CPUs.
- Latencia: Al implementar algoritmos directamente en hardware, se puede lograr una latencia más predecible y constante, lo cual es crucial en

aplicaciones de tiempo real o en sistemas criptográficos resistentes a ataques de *timing*.

- Seguridad: Las FPGAs pueden ofrecer una capa adicional de seguridad al implementar algoritmos criptográficos directamente en hardware, dificultando ciertos tipos de ataques que podrían ser más factibles en sistemas basados en software. Dichos ataques se describirán en una sección posterior.

- No hay sobrecarga del sistema operativo: A diferencia de las CPUs, que generalmente ejecutan un sistema operativo, las FPGAs pueden dedicar todos sus recursos a la tarea específica sin la sobrecarga adicional que conlleva la ejecución de un software sobre un OS (capa de SW lógica, interpuesta encima del HW a emplear).

Sin embargo, es importante también considerar las ventajas de las CPUs:

- Flexibilidad: Las CPUs son más flexibles y pueden ejecutar una amplia variedad de tareas sin necesidad de rediseñar el hardware.

- Facilidad: El desarrollo de software para CPUs es generalmente más accesible y requiere menos conocimientos especializados que el diseño de hardware para FPGAs.

- Desarrollo software: Las CPUs cuentan con un amplio sistema de herramientas de desarrollo, bibliotecas y frameworks que pueden acelerar el desarrollo de aplicaciones.

- Coste: En muchos casos, las soluciones basadas en CPU pueden ser más económicas, especialmente para aplicaciones de menor escala.

La elección entre FPGA y CPU dependerá de los requisitos específicos del proyecto, incluyendo factores como rendimiento, consumo de energía, seguridad, costo y tiempo de desarrollo. En conclusión, a pesar de sus debilidades, las características de la FPGA la hacen favorable para el proyecto.

8.2 Diseño y arquitectura

En esta sección, se presenta la arquitectura general del sistema, detallando los diferentes módulos que lo componen y cómo interactúan entre sí.

8.2.1 Estructura de Alto nivel

En una visión general se puede dividir el sistema en dos partes o componentes:

- La implementación hardware en la FPGA que actuará como usuario A: Este componente se encarga de implementar los algoritmos criptográficos directamente en hardware, ofreciendo alto rendimiento y seguridad.
- El desarrollo software sobre un PC que actuará como usuario B: Este componente implementa los mismos algoritmos en software, permitiendo una comparación directa con la implementación hardware.

Ambas partes del sistema generan su propio par de claves para luego comunicarse entre sí a través del protocolo UART (*Universal Asynchronous Receiver-Transmitter*, Transmisor-Receptor Asíncrono Universal) [98] y establecer una comunicación segura usando ECC.

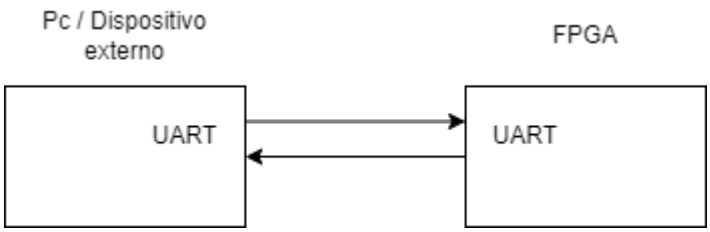

Figura 34: Diseño del criptosistema

La Figura 34 ilustra la estructura de alto nivel del sistema, mostrando cómo los dos módulos se comunican entre sí a través del protocolo UART, representado con flechas. Esta comunicación es fundamental para el intercambio de claves y la ejecución de los algoritmos criptográficos en ambos componentes

8.2.2 Componentes del sistema y módulos

Los componentes PC y FPGA realizan las mismas acciones, ya que el objetivo es tener una comunicación entre ambos usando curvas elípticas. Como se ha detallado con anterioridad, se ha desarrollado una implementación pura sin hacer uso de librerías desarrolladas específicamente para usos criptográficos que facilitan la implementación en este componente.

El componente FPGA se encarga de generar el par de claves de A y comunicarse con B (el PC) a través del protocolo UART. Está compuesto por cuatro módulos fundamentales:

- Módulo ECCDH: Es el módulo encargado de realizar el intercambio de claves. Este módulo implementa operaciones criptográficas como la suma de puntos de la curva y la multiplicación de un punto por un escalar. Genera de forma aleatoria la clave privada de A y evalúa que los puntos generados por B pertenezcan a la curva.

  En resumen, se encarga de realizar la implementación de la curva elíptica elegida y de realizar las validaciones.

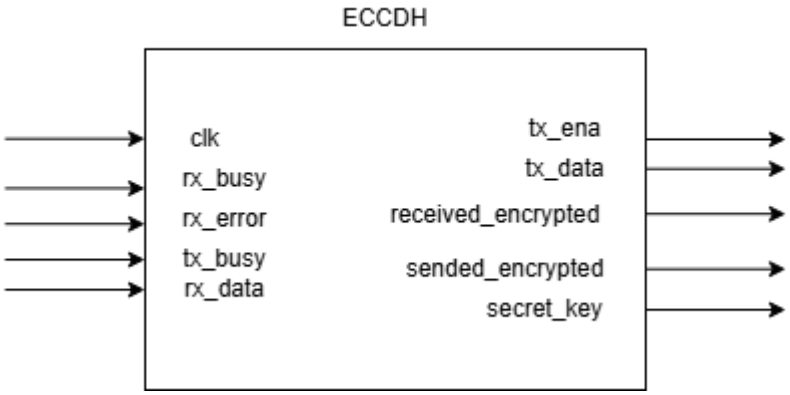

Figura 35: Diseño del módulo ECCDH

  La Figura 35 muestra el diseño del módulo ECCDH con las respectivas entradas, siendo estas las siguientes:

  - *clk* (*clock*): señal de reloj para la sincronización
  - *tx_busy:* Indica que el módulo está ocupado transmitiendo datos.
  - *rx_error*: Señal de error en la recepción de datos.
  - *tx_busy:* Indica que el módulo está ocupado transmitiendo datos.
  - *rx_data:* Datos recibidos.
  - *tx_ena:* Habilita la transmisión de datos.
  - *tx_da*ta: Datos a ser transmitidos.
  - *received_encrypted:* Indica que se ha recibido un mensaje cifrado.
  - *sended_encrypted:* Indica que se ha enviado un mensaje cifrado.
  - *secret_key:* Clave secreta generada tras el intercambio.

- Módulo AES: Este módulo realiza el encriptado y desencriptado de mensajes entre la FPGA y el dispositivo externo. Previo a esto el módulo ha recibido de ECCDH la llave para cifrar y descifrar.

La imagen a continuación muestra el diseño del módulo AES, que incluye las siguientes entradas y salidas:

- *clk:* Señal de reloj para sincronización.
- *key:* Llave utilizada para el cifrado y descifrado de datos.
- *cypher_to_decrypt:* Datos cifrados que se desean descifrar.
- *received:* Indica que se ha recibido un mensaje cifrado para ser procesado.
- *encrypted_data:* Datos resultantes tras el cifrado.

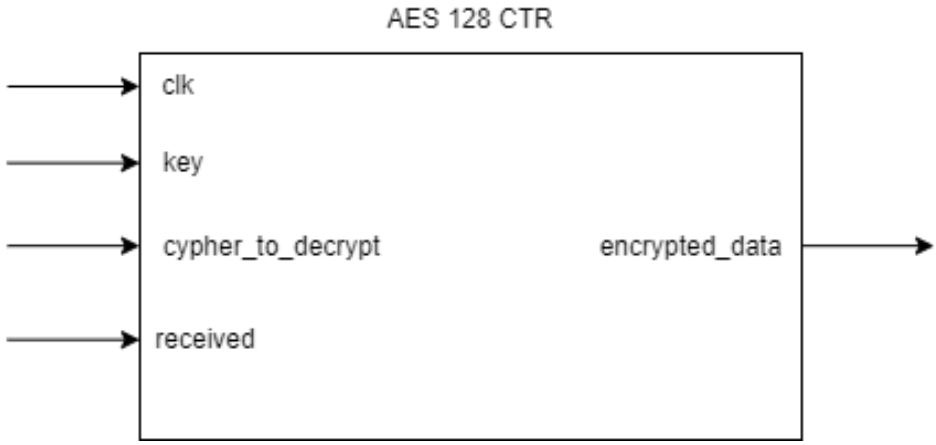

Figura 36: *Diseño del módulo AES*

- Módulo UART: Es el módulo encargado de la comunicación de la FPGA con cualquier otro dispositivo a través de este protocolo.

  El protocolo se ha personalizado para que reciba los datos en forma de bits que luego procesa en transmite al componente ECCDH en paquetes de 64 bits.

  A la hora transmitir los datos, el componente realizará el proceso inverso. Recibirá en paquetes de 64 bits la clave pública generada por el módulo ECCDH y luego transmitido bit a bit al componente externo. Hay que mencionar que este módulo ha sido facilitado por TechForum de DigiKey [99].

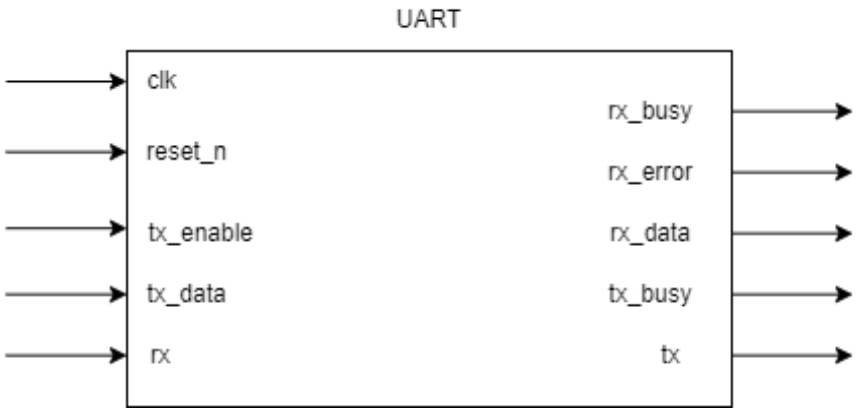

Figura 37: *Diseño del módulo UART*

La Figura 37 muestra el diseño del módulo UART, que incluye:

- *reset_n:* Señal de reinicio activa baja.
- *tx_enable:* Habilita la transmisión de datos.
- *tx_data*: Datos a ser transmitidos.
- *rx*: Datos recibidos desde el dispositivo externo.
- *rx_busy*: Indica que el módulo está ocupado recibiendo datos.
- *rx_error:* Indica si ha habido un error en la recepción de datos.
- *rx_data:* Datos recibidos y listos para ser procesados.
- *tx_busy:* Indica que el módulo está ocupado transmitiendo datos.
- *tx*: Datos transmitidos al dispositivo externo.

- Módulo de control: Módulo encargado de establecer las conexiones entre el módulo ECCDH y el módulo UART, recopilando los datos de cada uno de los módulos y “almacenando” en unas señales internas dichos datos. Estas señales son enviadas al resto de módulos que las requieran para su tratamiento.

  Las señales son las que permiten el correcto funcionamiento entre los módulos de los que se compone el proyecto, como se puede apreciar en la siguiente imagen.

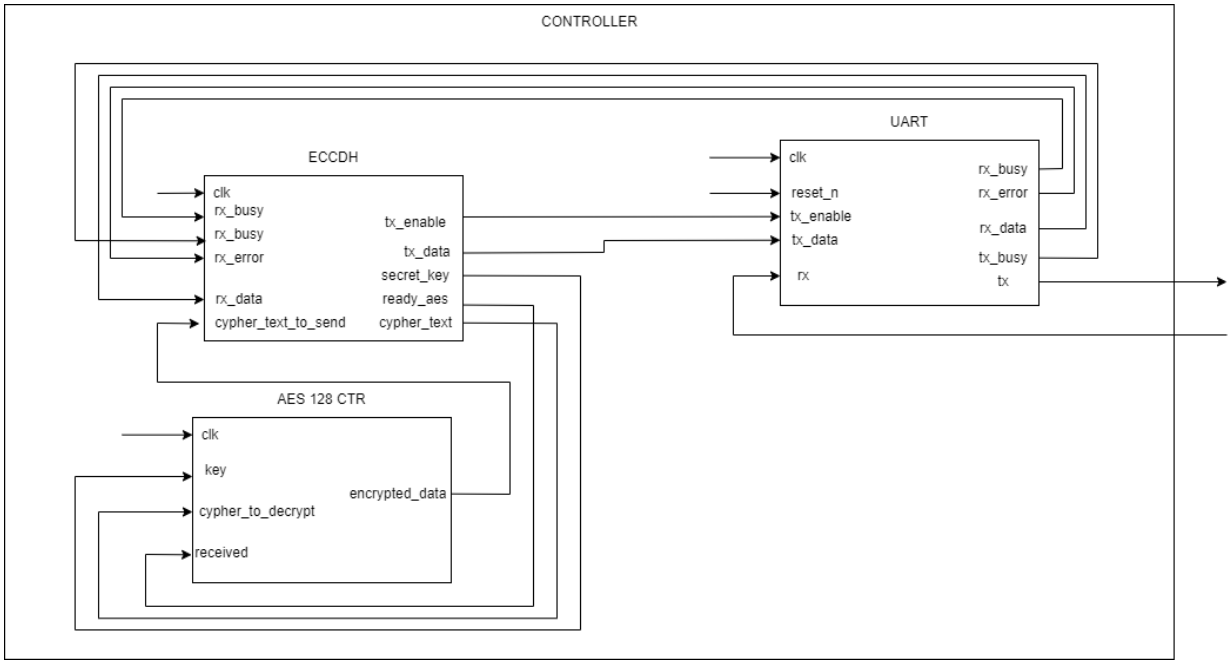

Figura 38: *Diseño del módulo de control*

Un aspecto destacable de la arquitectura es la centralización de las comunicaciones en el módulo ECCDH. Este módulo no solo maneja el intercambio de claves públicas, sino que también se encarga de enviar la clave de sesión al módulo AES y de transmitir los mensajes cifrados a través de la UART.

8.3 Optimización

La optimización es un aspecto crítico en la implementación en hardware. Aquí se discutirán las diversas técnicas de optimización empleadas en el diseño, así como las posibles mejoras que podrían realizarse en futuras iteraciones.

8.3.1 Mejora sobre hardware

Aunque no se han implementado mejoras específicas respecto a versiones anteriores del proyecto, se han identificado varias áreas potenciales de optimización:

- Implementación de *pipeline*: La introducción de técnicas de *pipeline* en las operaciones aritméticas de la curva elíptica podría mejorar significativamente el rendimiento. Por ejemplo, la operación de suma de puntos podría dividirse en cinco etapas:
    1. En la primera etapa, se haría una inicialización y comprobación de casos especiales como P = O ó Q = O.

2. En la etapa segunda, realiza el cálculo de la pendiente S evaluando previamente si P = Q o P ≠ Q.
3. La tercera etapa cubre el cálculo de la coordenada x del resultado.
4. Para la etapa cuatro se calcula la coordenada y, siendo la última la que devuelve el resultado.

Esta técnica permitiría un mayor *throughput* al procesar múltiples operaciones simultáneamente.

- Mayor modularidad: El módulo ECCDH podría beneficiarse de una mayor descomposición en submódulos. Esto implicaría crear módulos separados para operaciones como: suma de puntos, multiplicación por escalar, validación de puntos y generación de claves.

  De manera similar, el módulo AES podría dividirse en submódulos para cada etapa del algoritmo (AddRoundKey, SubBytes, ShiftRows, MixColumns).

  Esta modularización mejoraría la mantenibilidad, reutilización, testeabilidad y escalabilidad del sistema, aplicando los principios SOLID [100] al diseño de hardware.

- Mejoras de seguridad: Se podrían implementar varias técnicas para aumentar la seguridad del sistema:
  - Ecualización de tiempos de ejecución: Asegurar que todas las operaciones consuman la misma cantidad de tiempo para proteger contra ataques de *timing*.
  - Protección del *bitstream* (una secuencia de bits que configura la lógica interna de la FPGA): Utilizar ficheros *bitstream* cifrados y configurar la FPGA para prevenir lecturas no autorizadas.
  - Generación de números aleatorios: Implementar un *True Random Number Generato*r (TRNG) basado en hardware para mejorar la calidad de las claves privadas generadas.

- Optimización de recursos: Se podría realizar un análisis detallado del uso de recursos de la FPGA para identificar áreas de mejora. Esto podría incluir:
  - Optimización de las implementaciones de operaciones aritméticas para reducir el uso de LUTs y DSPs.

    - Uso eficiente de la memoria interna de la FPGA para reducir la latencia de acceso a datos.
    - Implementación de técnicas de reconfiguración dinámica para maximizar el uso de recursos disponibles.

- Paralelización: Explorar oportunidades de paralelización adicionales, como:
    - Implementación de múltiples instancias de operaciones de curva elíptica para procesar varias transacciones simultáneamente.
    - Paralelización de las rondas de AES para aumentar el *throughput* de cifrado/descifrado.

- Optimización de la interfaz UART: Considerar la implementación de *buffers* (almacenamiento intermedio temporal de información) más eficientes o incluso explorar el uso de interfaces de mayor velocidad como PCIe para aplicaciones que requieren mayor ancho de banda.

- Implementación de contramedidas contra ataques de canal lateral: Además de la ecualización de tiempos, se podrían implementar técnicas como:
    - Enmascaramiento de datos para proteger contra análisis de consumo de energía.
    - Introducción de operaciones *dummy* (operaciones ficticias que no afectan el resultado final) para dificultar el análisis de timing.
    - Implementación de técnicas de *shuffling* (aleatorización en el orden de ejecución de ciertas operaciones) para randomizar el orden de ejecución de ciertas operaciones.

- Mejora del generador de números aleatorios: Además del TRNG mencionado, se podría considerar la implementación de un generador de números pseudoaleatorios (PRNG) basado en hardware para mejorar la eficiencia en la generación de valores aleatorios necesarios para el protocolo ECDH.

- Optimización de la curva elíptica: Explorar la implementación de operaciones de curva elíptica utilizando coordenadas proyectivas o jacobianas, que pueden ofrecer mejor rendimiento que las coordenadas afines en ciertas operaciones.

La implementación de estas mejoras podría resultar en un sistema más eficiente, seguro y flexible, capaz de manejar una mayor variedad de aplicaciones criptográficas.

8.4 Validación y pruebas

En esta sección, se detallarán los diversos tests y procedimientos de validación aplicados a el sistema, desde pruebas unitarias de cada módulo hasta pruebas de integración del sistema completo. También se analizarán los errores encontrados durante el proceso de desarrollo y cómo fueron abordados y resueltos.

8.4.1 Test Unitarios. Pruebas por Módulos.

1. Módulo ECCDH: Para realizar las pruebas de este módulo se ha tenido que comprobar de forma independiente cada función que realiza las operaciones criptográficas:
   - Función 'xgcd' (módulo inverso): Se realizaron pruebas con valores pequeños y grandes para verificar la correcta implementación del algoritmo extendido de Euclides [101]. Como se muestra en las Figuras 39, 40, 41 y 42, el resultado obtenido concuerda con el esperado, validando el funcionamiento de la función.
     - ➢ Ejemplo 1:
       - Entrada: n = 847, p = 398
       - Resultado esperado: 359 (hexadecimal: 167)
       - Resultado obtenido: 359 (hexadecimal: 167)

```
pow inv n mod p: (en hex) 0x167
```

Figura 39: *Prueba en Python 3 usando xgcd*

testing_value1[255:0]	000000000000000000000000000000000000000000000000000000000000018e
testing_value2[259:0]	000000000000000000000000000000000000000000000000000000000000034f
xgcd_testing[255:0]	0000000000000000000000000000000000000000000000000000000000000167

Figura 40: *Prueba en Vivado usando la implementación xgcd*

Con todos los bits del número a calcular a '1', es decir, 256 bits a '1' lógico, y cuando como modelo el parámetro P de la curva secp256r1 se han obtenido los valores esperados.

- ➢ Ejemplo 2 (valores de 256 bits):
  - Entrada: n = 2^256 - 1, p = parámetro P de la curva secp256r1

```
pow inv n mod p: (en hex) 0xe1e1e1e10f0f0f0f69696969b4b4b4b45a5a5a5b0f0f0f0e87878787c3c3c3c2
```

Figura 41: *Prueba en Python 3 usando xgcd para 256bits*

xgcd_testing[255:0]	e1e1e1e10f0f0f0f69696969b4b4b4b45a5a5a5b0f0f0f0e87878787c3c3c3c2

Figura 42: *Prueba en Vivado usando la implementación xgcd para 256bits*

- Suma de puntos: Se probaron cuatro casos fundamentales: P = O, Q = O, P = Q, y P ≠ Q. Para simplificar, se utilizó el punto base generador G de la curva secp256r1 en varias pruebas.
  - ➢ Caso P = O, Q = G:
    - Resultado esperado: G
    - Resultado obtenido: G

```
add_points with P = (0,0) and Q = G:  0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
, 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5
```

Figura 43: *Prueba con P = O, Q= G en Python*

testing_value_x[255:0]	0000000000000000000000000000000000000000000000000000000000000000
testing_value_y[255:0]	0000000000000000000000000000000000000000000000000000000000000000
add_point_testing[511:0]	6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c2964fe3
add_point_testing_x[255:0]	6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
add_point_testing_y[255:0]	4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5

Figura 44: Prueba con *P = O, Q= G en Vivado*

Las Figuras 43 y 44, reflejan el correcto funcionamiento del método para el caso mencionado. Se obtendrá el mismo resultado para P = G y Q = O por este motivo se omitirá.

- ➢ Caso P = G, Q = G:
  - Resultado esperado: [2]G (duplicación de punto)
  - Resultado obtenido: [Coordenadas de 2G]

Como se observa en las Figuras 45 y 46, el resultado obtenido coincide con las coordenadas esperadas para 2*G, confirmando la precisión de la función en este escenario.

```
add_points with P and Q = G:  0x7cf27b188d034f7e8a52380304b51ac3c08969e277f21b35a60b48fc47669978,
0x7775510db8ed040293d9ac69f7430dbba7dade63ce982299e04b79d227873d1
```

Figura 45: *Prueba con P y Q = G en Python*

add_point_testing_x[255:0]	7cf27b188d034f7e8a52380304b51ac3c08969e277f21b35a60b48fc47669978
add_point_testing_y[255:0]	07775510db8ed040293d9ac69f7430dbba7dade63ce982299e04b79d227873d1

Figura 46: *Prueba con P y Q = G en Vivado*

➢ Caso P ≠ Q:

- P=(7b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296,6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296)
- Q = G
- Resultado: [Coordenadas del punto resultante]

Las Figuras 47 y 48 muestra el resultado del cálculo y las coordenadas obtenidas para el punto resultante, confirmando el correcto funcionamiento de la operación de suma

```
add_points with P and Q different:  0x1663451aa72afc6bc56e7c4fd2499daaaa34d58badfa99439d517ab82bae9fe2,
0xaa5126873f8362d6dc4641423e11dc576f48763f6fdf15c0e7fc500d8bd8e61f
```

Figura 47: *Prueba con Q = G y P = K en Python*

add_point_testing_x[255:0]	1663451aa72afc6bc56e7c4fd2499daaaa34d58badfa99439d517ab82bae9fe2
add_point_testing_y[255:0]	aa5126873f8362d6dc4641423e11dc576f48763f6fdf15c0e7fc500d8bd8e61f

Figura 48: *Prueba con Q = G y P = K en Vivado*

- Multiplicación escalar: Se probó la multiplicación de un punto por un escalar, utilizando tanto valores pequeños, Figuras 49 y 50, como grandes para el escalar, Figuras 51 y 52.

➢ Ejemplo 1:

- Escalar: A1452bce7f12
- Punto: G (generador de la curva)
- Resultado: [Coordenadas del punto resultante]

```
multiply_point_by_scalar: 0x6f63da954131136ba1c3e7e66c4bd71b3e676b8e8d73d726165d0b0095fcd83b,
0x1ce9af77befd4b9b7eca331023ba3cd88f432fde91ad86c43db903d3cbdb5bbe
```

Figura 49: *Prueba multiplicación punto por escalar en Python*

pointXscalar_testing_x[255:0]	6f63da954131136ba1c3e7e66c4bd71b3e676b8e8d73d726165d0b0095fcd83b
pointXscalar_testing_y[255:0]	1ce9af77befd4b9b7eca331023ba3cd88f432fde91ad86c43db903d3cbdb5bbe

Figura 50: *Prueba multiplicación punto por escalar en Vivado*

➢ Ejemplo 2 (escalar de 256 bits):

- Escalar: 100bff25f3e2032e807afc05b3e36c1e7a1378b74fbe44c575f89c7a987b8d3b
- Punto: G
- Resultado: [Coordenadas del punto resultante]

```
SRL =  0x100bff25f3e2032e807afc05b3e36c1e7a1378b74fbe44c575f89c7a987b8d3b
multiply_point_by_scalar: 0x2bffd552411e95ea21e3be67ad7c148b6a54a8e7de5ee9f398c04f00b6d5b1a2,
0xf90cf8df8aa7906bfb1e7bec4504371ec5aeeb35e37aad4d48e88055c630b204
```

Figura 51: *Prueba de multiplicación punto p escalar con 256 bits en Python*

testing_value_x[255:0]	100bff25f3e2032e807afc05b3e36c1e7a1378b74fbe44c575f89c7a987b8d3b
pointXscalar_testing[511:0]	2bffd552411e95ea21e3be67ad7c148b6a54a8e7de5ee9f398c04f00b6d5b1a2f90
pointXscalar_testing_x[255:0]	2bffd552411e95ea21e3be67ad7c148b6a54a8e7de5ee9f398c04f00b6d5b1a2
pointXscalar_testing_y[255:0]	f90cf8df8aa7906bfb1e7bec4504371ec5aeeb35e37aad4d48e88055c630b204

Figura 52: *Prueba de multiplicación punto por escalar con 256 bits en Vivado*

- Generación de clave privada: Se implementó y probó una función para generar claves privadas de 256 bits, asegurando que el valor generado estuviera en el rango adecuado (1 ≤ d < n, donde n es el orden del punto generador).

  En la Figura 53 se observa el resultado de esta prueba, donde se valida que cada clave generada cumple con las condiciones establecidas para el algoritmo.

testing_random[255:0]	15ad1f0e7ffffff2965f2fb09753296554518e57ffffff7ffffff80000000

Figura 53: *Prueba de generación de clave privada d, 1 ≦ d ≦ n-1*

- Validación de puntos: Se probó la función de validación de puntos para asegurar que detectara correctamente puntos que pertenecen, como se puede observar en la Figuras 54 y 55, y los puntos que no pertenecen a la curva, Figura 56 y 57.

  ➢ Ejemplo 1:

- P=(f963cc7059965a0dec32dd6764ae36647ee1e72a00ab022f7c10ee432db78ad, 7525e67c0a18fcab21926d6a1f137311234793ab69ef7f12babc3363cf74d32e)

```
publica de FPGA:  0xbf963cc7059965a0dec32dd6764ae36647ee1e72a00ab022f7c10ee432db78ad
0x7525e67c0a18fcab21926d6a1f137311234793ab69ef7f12babc3363cf74d32e
Clave pública de la FPGA esta en la curva? True
```

Figura 54: *Prueba de validación de puntos en Python*

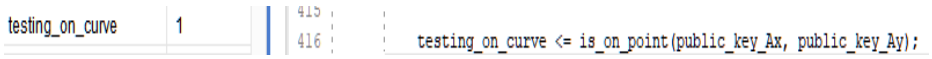

Figura 55*: Prueba de validación de puntos en Vivado*

➢ Ejemplo 2:

- P=(6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d8aa7972, 8b5d90bf0c91e1fab33a0ae814aeffb946e76f97d8faec8b7a4cf33ac892c01d)

```
publica de FPGA:  0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d8aa7972
0x8b5d90bf0c91e1fab33a0ae814aeffb946e76f97d8faec8b7a4cf33ac892c01d
Clave pública de la FPGA esta en la curva? False
```

Figura 56: *Prueba de validación de puntos con P fuera de la curva en Python*

testing_on_curve	0
testing_a[255:0]	6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d8aa7972
testing_b[255:0]	8b5d90bf0c91e1fab33a0ae814aeffb946e76f97d8faec8b7a4cf33ac892c01d

Figura 57: *Prueba de validación de puntos con P fuera de la curva en Vivado*

2. Módulo AES: Se realizaron pruebas de cifrado y descifrado utilizando vectores de prueba conocidos:

   Key = 2b7e151628aed2a6abf7158809cf4f3c
   Texto plano = 6bc1bee22e409f96e93d7e117393172a
   Nonce = f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff

recived_encrypted[127:0]	874d6191b620e3261bef68649
is_recived	1
sended_encrypted[127:0]	874d6191b620e3261bef68649
waiting_next	1
has_encrypted	1
to_decrypt[127:0]	6bc1bee22e409f96e93d7e117
testing_plaintext[127:0]	6bc1bee22e409f96e93d7e11739
testing_encrypted[127:0]	874d6191b620e3261bef68649
testing_decrypted[127:0]	6bc1bee22e409f96e93d7e117
testing_key[127:0]	2b7e151628aed2a6abf715880

Figura 58: *Prueba del módulo AES*

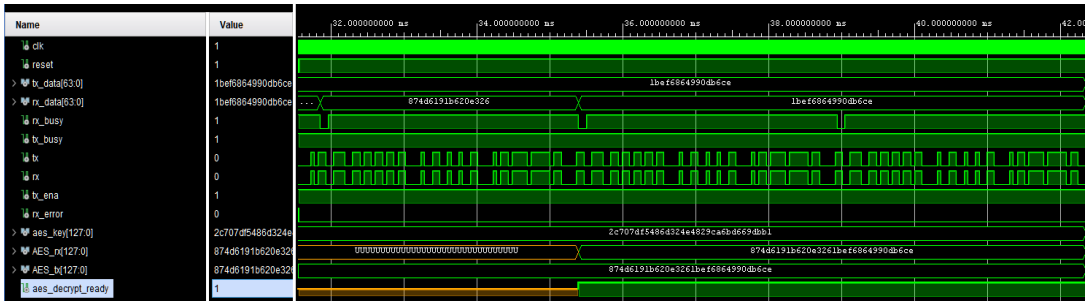

Figura 59: *Prueba del módulo AES transmitiendo mensajes*

Las Figuras anteriores reflejan el correcto funcionamiento del módulo AES. Aunque se usan datos de pruebas, el módulo encripta el texto plano y lo transmite por la UART cuando esta se encuentra disponible (AES_tx). Por cómo se ha decidido implementar la simulación, el mensaje recibido del dispositivo externo es el mismo que el enviado. Observando la imagen AES_rx recibe el dato a desencriptar cuando ra_data, manejado por el módulo ECCDH, recibe el mensaje a su totalidad (128 bits) poniendo la señal aes_decrypt_ready a '1', indicando que el módulo puede iniciar el desencriptado.

Se verificó que el texto cifrado producido coincidiera con el esperado y que el proceso de descifrado recupera correctamente el texto original.

3. Módulo de comunicación UART: Este módulo, facilitado por DigiKey, cuenta con su propio banco de pruebas (*testbench*) que demuestra el correcto funcionamiento del protocolo UART. Se verificaron aspectos como:
    - Correcta transmisión y recepción de bytes.
    - Manejo adecuado de las señales de control (start bit, stop bit).
    - Funcionamiento a diferentes velocidades de baudios.

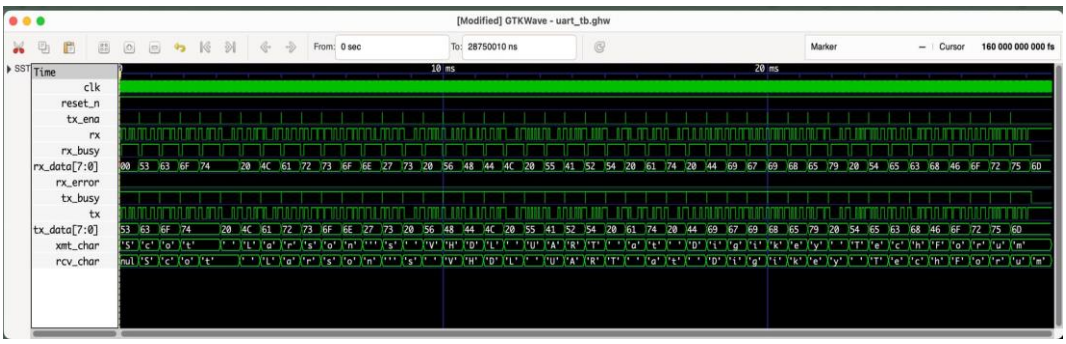

Figura 60: *Prueba de funcionamiento de la UART (Fuente DigiKey TechForum)*

La imagen demuestra cada uno de los puntos anteriores, validando la precisión y eficiencia del módulo en diversas condiciones.

4. Módulo de control: Para este módulo, se realizaron pruebas simulando la entrada de datos (clave pública de B) y verificando la correcta interacción entre la UART y el módulo ECCDH. Se prestó especial atención a:
    - Correcto enrutamiento de datos entre módulos.
    - Sincronización adecuada de las operaciones.
    - Manejo apropiado de los estados del sistema durante el proceso de establecimiento de claves.

    Las Figuras 61 y 62, ilustran cada uno de estos puntos, confirmando la precisión y robustez del módulo de control en su función dentro del sistema.

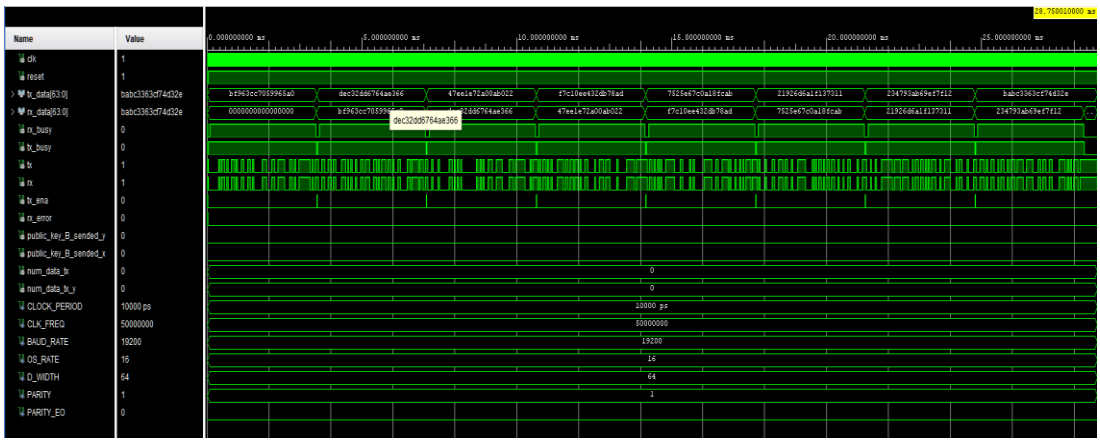

Figura 61*: Prueba del módulo de control (1)*

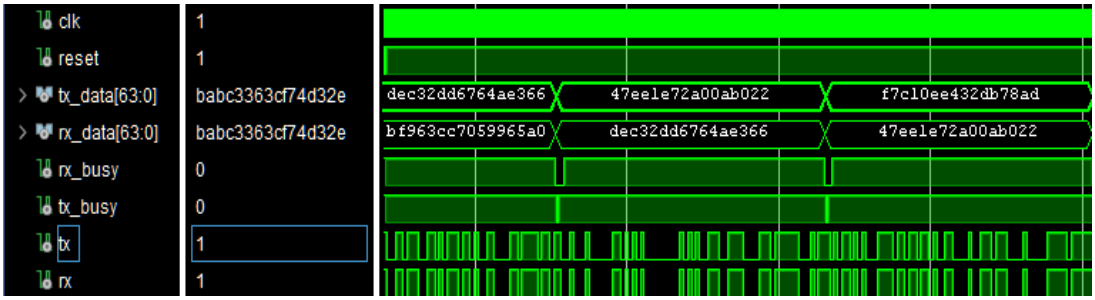

Figura 62: *Prueba del módulo de control (2)*

8.4.2 Pruebas de rendimiento

- Curvas elípticas: Para simular el comportamiento del dispositivo durante la generación del par de claves, las instrucciones encargadas de producir la clave pública y privada se han encapsulado en un mismo bloque.

  En la Figura 63, se muestra el estado del módulo ECCDH en t = 0 ms. Las señales clk, start, key_generated (que indica si se ha generado la clave) y aux_key (que representa la clave pública) se encuentran inactivas en este momento, sin haber iniciado el proceso de generación de claves.

  La Figura 64 muestra cómo la señal *start* se activa en t = 10 ms, marcando el inicio de la generación de claves. Como resultado de este inicio, en t = 10.05 ms, la clave pública ya ha sido calculada y su valor se refleja en la señal aux_key, mientras que el flag key_generated cambia a '1', indicando que el proceso de generación de claves se ha completado.

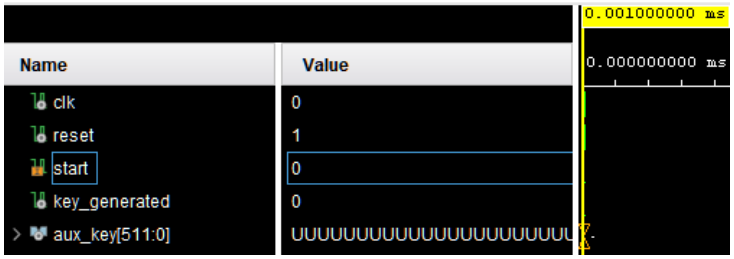

Figura 63: Prueba de rendimiento del módulo ECDH en vivado (1)

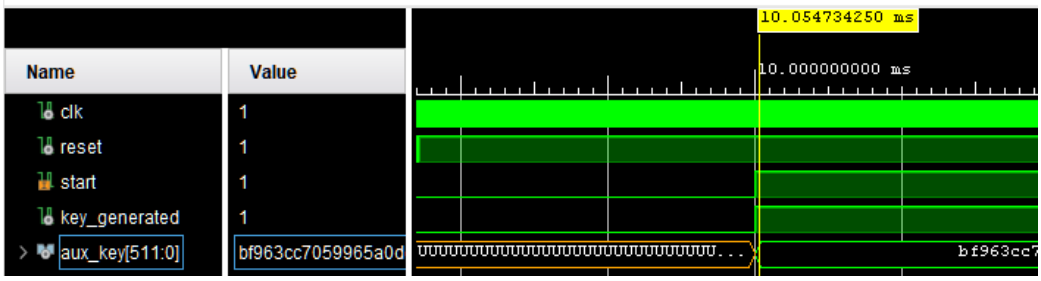

Figura 64: *Prueba de rendimiento del módulo ECDH en vivado (2)*

- AES: Siguiendo el mismo esquema realizado para las curvas elípticas, la simulación del tiempo de ejecución en el algoritmo AES en la FPGA se inicia a través de una señal de activación (*start*). Esta señal se pone en '1' lógico tras 10 ms, iniciando así el proceso de cifrado.

  Para analizar los resultados, se observan las dos imágenes mostradas a continuación, en las que se destacan tres señales clave: start_AES (inicia el proceso de cifrado), has_encrypted (indica que el plaintext ha sido cifrado) y testing_encrypted (contiene el valor del texto cifrado).

  En la Figura 65, correspondiente al tiempo inicial de la simulación (t = 0 ms), se observa que las señales clk, start, has_encrypted y testing_encrypted no tienen un valor activo, ya que aún no se ha iniciado el proceso de cifrado. Mientras, la Figura 66 muestra el instante en t = 10 ms en el cual start_AES se activa al ponerse en '1', marcando el inicio del cifrado. Poco después, en t = 10.03 ms, la señal has_encrypted toma el valor '1', lo que indica que el cifrado se ha completado exitosamente. Al mismo tiempo, testing_encrypted refleja el valor del texto cifrado obtenido.

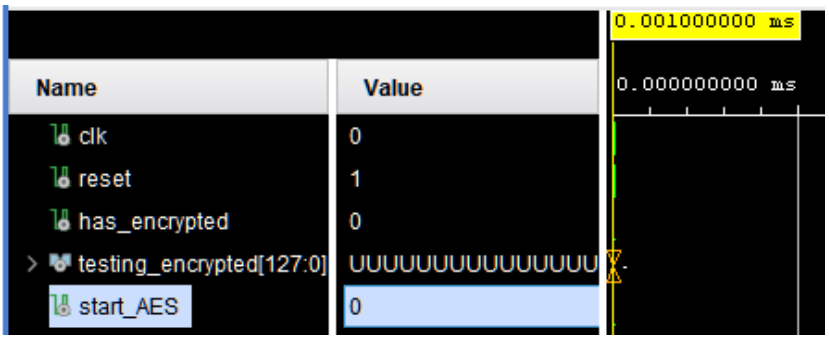

Figura 65: *Prueba de rendimiento del módulo AES en vivado (1)*

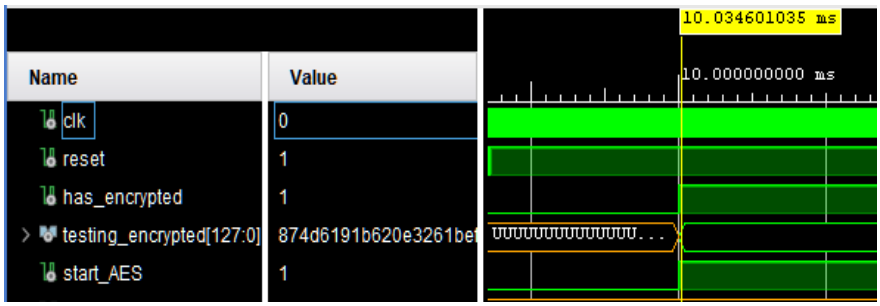

Figura 66: *Prueba de rendimiento del módulo AES(2)*

En simulaciones RTL (*Register Transfer Level*) como las realizadas en Vivado, los tiempos de las operaciones pueden reflejar el comportamiento lógico del hardware, aunque los tiempos reales dependen de la síntesis final y la implementación física. En la implementación del proyecto, al activar la señal de inicio en t = 10 ms, los resultados

de operaciones complejas como la generación de claves y el cifrado AES se obtienen en un corto margen de tiempo (t = 10.05 ms).

Cabe mencionar que algunas operaciones, como la generación de claves, se realizan de forma combinacional fuera de los bloques *process*. Esto significa que se evalúan de manera puramente lógica sin estar sincronizadas al reloj, lo que permite que se completen en un solo ciclo de simulación. Sin embargo, es importante notar que en la implementación física, estas operaciones combinacionales tendrán un retardo de propagación que determinará la frecuencia máxima de operación del circuito.

8.4.3 Análisis de errores

En esta sección se analizarán los errores fundamentales que se han encontrado a la hora de implementar los algoritmos en la FPGA.

Primero se hablará de problemas de diseño:

a) Uso de tipo de datos erróneos. Durante la implementación inicial de los algoritmos en VHDL, se optó por utilizar el tipo de datos std_logic_vector, pensando en la futura transmisión bit a bit por la UART. Sin embargo, este enfoque causó problemas en las operaciones, ya que se requerían conversiones constantes a tipo *unsigned,* el cual se utilizó posteriormente.

   El cambio al tipo *unsigned* tampoco solucionó el problema y, de hecho, introdujo nuevos desafíos, especialmente en las operaciones criptográficas. En ciertos momentos, los valores almacenados podían ser negativos, y *unsigned* no puede representar valores negativos, lo que conducía a errores y valores incorrectos en los puntos de la curva elíptica.

   Para resolver estos problemas, se decidió finalmente utilizar el tipo de datos *signed* en las operaciones criptográficas. Este tipo de datos permitió manejar valores negativos correctamente, manteniendo la precisión y exactitud necesarias en las operaciones.

b) Problemas en la longitud de los datos: Las operaciones como la suma y, especialmente, la multiplicación, requerían la verificación de la longitud de los resultados para evitar desbordamientos. Para abordar este problema de manera eficiente en la simulación, se implementó una solución que utiliza la función resize() de VHDL. Esta función permite ajustar temporalmente el

tamaño de las variables durante la simulación, facilitando el manejo de datos de diferentes longitudes según la operación realizada.

El uso de resize() fue fundamental para determinar el tamaño adecuado de las variables en la simulación. Basándose en estas observaciones, se definieron de manera fija las longitudes de las variables en el código VHDL para la implementación en hardware, dado que en un entorno real no es posible ajustar dinámicamente el tamaño de las variables. Este fenómeno se conoce como "deformación de hardware".

c) Problemas en la suma de puntos. Durante la implementación de la operación de suma de puntos en la criptografía basada en curvas elípticas, surgen inicialmente problemas relacionados con la longitud de los datos. Una vez solucionado este aspecto, surgió otro desafío: durante las pruebas, los resultados no eran los esperados cuando se sumaban dos puntos distintos $P \neq Q$.

   El problema se originó porque, en la determinación de la pendiente m, algunas operaciones internas pueden producir valores negativos, especialmente si $P > Q$. No se había considerado este caso específico en la implementación, lo que provocó resultados incorrectos.

   La solución, después de identificar el error, fue comprobar si el resultado de la operación era negativo y, en ese caso, aplicar el módulo sobre el cuerpo p de la curva. Este ajuste permitió manejar adecuadamente los valores negativos, garantizando que la suma de puntos produjera los resultados correctos conforme a las reglas de la aritmética de curvas elípticas.

d) Generación de clave privada. Durante la simulación, se observan problemas en la generación de la clave privada, ya que constantemente se obtenía el mismo valor. Tras investigar, se descubrió que esto se debía a que la función responsable de generar números aleatorios para la clave utilizaba los mismos valores de *seed1* y *seed2* en el método *uniform()*, lo que resultaba en claves repetitivas. La solución, encontrada con la ayuda de la comunidad de StackOverflow [102], fue implementar el generador de números aleatorios dentro de un bloque *process*. Este enfoque permitió generar valores únicos en cada ejecución.

Otro desafío fue la imposibilidad de generar un valor aleatorio de 256 bits directamente. Para abordar esto, se dividió la generación del valor aleatorio en 8 partes de 32 bits. Sin embargo, para garantizar que el valor generado estuviera dentro del rango del cuerpo p de la curva, se aplica el módulo p al final del proceso. Esta solución aseguró que las claves privadas fueran adecuadamente distribuidas dentro del espacio definido por el cuerpo p.

Errores en el rendimiento: Inicialmente se establecieron todas las variables a 768 bits y se realizó un cambio de tamaño para asegurar de mantener siempre la longitud. Esta solución que resolvió el problema de la longitud de la clave introducía una gran demora al realizar las simulación. Posteriormente se aplicó la solución anterior de fijar ciertas longitudes para los casos necesarios.

En cuanto a los errores de implementación:

a) Anidación secuencial**:** Tras implementar el sistema en VHDL y verificar el funcionamiento de todos los módulos, surgieron errores inesperados durante la síntesis. El problema se originó en los procesos del módulo ECCDH encargados de leer y enviar claves públicas, ya que VHDL permite procesos secuenciales anidados (*clocked process*) sólo en simulación, no en síntesis. La solución fue eliminar la anidación de procesos y reemplazarlos con máquinas de estado para gestionar las fases, lo que permitió completar la síntesis del proyecto correctamente.

b) Limitación de bucles: Durante la síntesis, también se identificaron otro problema: VHDL detectó un bucle sin una condición de salida clara y definitiva en el método xgcd, encargado de calcular el inverso. Este bucle '*while*' se detiene cuando el resto de la división es cero, pero el entorno de síntesis no interpretó esta condición como suficiente. Para resolverlo, se remplazó el bucle '*while*' con un bucle '*for*', limitando el número de iteraciones y añadiendo una condición de salida adicional para asegurar que el bucle se detenga cuando el resto de la división sea cero.

A pesar de haber solucionado los problemas anteriores y de lograr una simulación correcta, se experimentó un fallo durante la generación del *bitstream*, que se manifestó como un error TCLStackFree. Este error indica un problema en la gestión de memoria dentro del entorno de diseño, lo que puede suceder cuando se intenta liberar recursos que ya han sido liberados o cuando hay un desbordamiento de la pila

de llamadas. Esta situación ha dificultado la identificación de la causa raíz del problema y requiere investigación adicional para resolverlo adecuadamente.

Estos errores y sus soluciones proporcionaron valiosas lecciones sobre las complejidades de la implementación de algoritmos criptográficos en hardware, especialmente en lo que respecta a la precisión numérica, el manejo de datos y las limitaciones de las herramientas de síntesis.

9. Análisis de seguridad y ataques potenciales

En esta sección se realizará un análisis exhaustivo de la seguridad de la implementación, se identificarán los ataques potenciales y explicará cómo protegerse de ellos. Es crucial entender estas vulnerabilidades y amenazas para desarrollar sistemas criptográficos robustos y seguros.

9.1 Ataques contra AES

Se empezará explorando ataques contra el algoritmo AES-128, enfatizando en ataques específicos contra el modo CTR, que es el que fue escogido para la implementación en hardware.

- Ataques de fuerza bruta: Este ataque consiste en probar todas las combinaciones posibles de claves hasta encontrar la correcta. Es teóricamente posible, pero impráctico para claves de longitud adecuada como la que se usa. Para poner esto en perspectiva, con una clave AES-128, hay 2^128 (aproximadamente 3.4 x 10^38) combinaciones posibles. Incluso si se pudiera probar un billón de claves por segundo, tomaría aproximadamente 10^21 años probar todas las combinaciones. Por tanto, la implementación no es vulnerable ante este ataque en la práctica.

- Ataque de criptoanálisis diferencial: En este ataque, se analiza las diferencias en el cifrado de múltiples textos planos y cómo se reflejan en los textos cifrados para deducir la clave de cifrado. AES está diseñado específicamente para resistir este tipo de ataques. La implementación, al seguir fielmente el estándar AES, hereda esta resistencia. El diseño de la S-box en AES, por ejemplo, está optimizado para maximizar la resistencia contra el criptoanálisis diferencial.

- Ataque de criptoanálisis lineal: Consiste en intentar encontrar aproximaciones lineales entre el *plaintext* y el texto cifrado, y luego, utilizar estas aproximaciones para deducir la clave. Del mismo modo que el ataque anterior, las buenas prácticas implementadas en el desarollo protegen contra esto. AES utiliza operaciones no lineales (como la S-box) que dificultan enormemente este tipo de análisis.

- Ataques de pre-imagen y colisión: Estos ataques buscan encontrar dos entradas diferentes que produzcan el mismo texto cifrado o encontrar una entrada que produzca un texto cifrado específico. Es importante mencionar que estos ataques son más

relevantes en funciones *hash*, pero pueden aplicarse a cifrados en bloque si se usan modos inseguros como ECB. En este caso, este modo fue descartado precisamente por estas vulnerabilidades. El uso de CTR mitiga significativamente estos riesgos al introducir un contador que varía para cada bloque.

- Ataques relacionados con la clave: Estos ataques tratan de explotar relaciones matemáticas entre varias claves utilizadas para cifrar diferentes textos planos. La mayor vulnerabilidad de AES en este caso es el uso inadecuado de la clave. Siempre que se utilicen claves aleatorias e independientes, la implementación debería ser segura contra estos ataques. Es crucial utilizar un generador de números aleatorios criptográficamente seguro para la generación de claves.

- Reutilización del Contador (*nonce*): Si el mismo contador se reutiliza con la misma clave, los textos cifrados correspondientes a los mismos bloques de texto plano serán idénticos. Esto permite la detección de patrones, y eventualmente, una deducción del contenido del mensaje. En la implementación, es crucial asegurar que cada combinación de clave y contador sea única. Una forma de lograrlo es utilizar un *nonce* aleatorio para cada mensaje y combinar este *nonce* con un contador incremental para cada bloque.

- Ataque de Flujo de Texto Claro Conocido: Si un atacante conoce parte del texto y su correspondiente texto cifrado, puede derivar el flujo de texto cifrado y utilizar este flujo para descifrar otras partes del mensaje bajo el mismo contador y clave. Esto permite al atacante obtener partes del mensaje original sin conocer la clave completa. Para mitigar este riesgo, además de evitar la reutilización del contador, es importante evitar utilizar un *plaintext* con información predecible o estructurada. En aplicaciones donde parte del *plaintext* es conocido o predecible, se pueden implementar técnicas de *padding* aleatorio o de ofuscación de datos.

- Ataques de Integridad: Este ataque se relaciona con el hecho de que CTR es un modo no autenticado. Como ya se explicó anteriormente, un atacante puede alterar los mensajes cifrados sin ser detectado. Esta es una vulnerabilidad inherente al modo CTR que no se puede solucionar directamente. La manera de mitigarlo sería implementando un modo autenticado como GCM (*Galois/Counter Mode*) o utilizando un MAC (*Message Authentication Code*) en conjunto con CTR. En la implementación, aunque no se ha incluido autenticación, es crucial advertir a los usuarios sobre esta limitación y recomendar el uso de mecanismos de autenticación adicionales cuando sea necesario.

- Ataque de análisis de Tráfico: Aunque CTR convierte AES en un cifrado de flujo, es posible para un atacante realizar un análisis de tráfico para deducir patrones de

comunicación, frecuencias, tamaños de mensajes, etc. Esto no compromete el contenido del mensaje, pero puede revelar información sensible sobre la naturaleza de la comunicación. La manera ideal de mitigar esta vulnerabilidad es implementar técnicas de ofuscación de tráfico como *padding* de mensajes o añadiendo ruido al tráfico de comunicación. En la implementación, se podría considerar añadir un *padding* aleatorio a los mensajes para que todos tengan el mismo tamaño, dificultando así el análisis de tráfico.

- Ataques por inyección de texto cifrado: Un atacante puede inyectar bloques de texto cifrado que se descifrarán como *plaintext* predecible. Esto puede producir una inyección de comandos o datos maliciosos en un sistema. Esto se soluciona de la misma forma que los ataques de integridad, utilizando cifrado con autenticación. Aunque la implementación actual de CTR no previene directamente este tipo de ataque, es crucial advertir a los usuarios sobre esta vulnerabilidad y recomendar la implementación de mecanismos de autenticación adicionales en las capas superiores de la aplicación.

9.2 Ataques contra ECDH

En secciones anteriores se mencionaron los errores de usuario, sin embargo, existen ataques directos usados por los atacantes para vulnerar el sistema. Es importante entender estos ataques para poder diseñar contramedidas efectivas.

9.2.1 Ataques al logaritmo discreto [103]

- Ataque por fuerza bruta. En el contexto de curvas elípticas, este tipo de ataques intenta obtener el valor de la clave secreta K hasta poder satisfacer la ecuación $Q = KP$. Este método de ataque resulta altamente ineficiente debido a que requiere una gran potencia de cómputo por parte del atacante. Para una curva elíptica con un orden de aproximadamente 2^256 (como las que se usan en la implementación), un ataque de fuerza bruta es computacionalmente inviable con la tecnología actual, aunque previsible en un futuro cercano.

- *Baby-Step Giant-Step* [104]: Este tipo de ataque, descubierto por el matemático Daniel Shanks en 1971, plantea un algoritmo eficiente para resolver el problema del logaritmo discreto en grupos abelianos, incluyendo así el grupo de puntos de una curva elíptica. Este tipo de ataque es efectivo debido a la elección de m, donde $m = \lceil \sqrt{n} \rceil$ siendo n el orden del grupo. Observando la ecuación del

logaritmo discreto $\alpha^x = \beta$ (usando otra notación), el *Baby-Step Giant-Step* pretende reescribir el valor de x de la siguiente forma:

$$x = im + j, \text{ donde } m = \lceil \sqrt{n} \rceil, \ 0 \leq i < m, \text{ y } 0 \leq j < m$$

Por tanto, la ecuación quedaría de la siguiente forma sustituyendo x:

$$\alpha^x = \beta \Longrightarrow \alpha^{im+j} = \beta \Longrightarrow \alpha^j = \beta(\alpha^{-m})^i$$

El algoritmo computa α^j un número de veces j, almacenando el valor obtenido para cada par (j, α^j). Posteriormente se computa $\beta(\alpha^{-m})$ elevando el resultado a i y comparándolo con el valor obtenido de α^j hasta encontrar una colisión. En el momento que se encuentra dicha colisión, se procede a resolver el problema del logaritmo discreto. La Figura 67 ayuda a visualizar este proceso, mostrando cómo se efectúa la búsqueda y comparación de valores hasta encontrar la colisión necesaria.

```
Baby-Step Giant-Step

Input: Group n Orden
Output: DLA solution

begin
    m=ceiling(n)
    for j in 0 ≤ j<m  then:
        D = Compute (j, a^j)
    S = Compute   β(a^-m)
    for  i in 0 ≤ i < m then:
        Q = Compute S^i
        if Q = D then:
            return i*m + j
        else
            S = S*a^-m

end
```

Figura 67: *Pseudo algoritmo del ataque Baby-Step Giant-Step*

Este algoritmo tiene una complejidad de O(√n), lo que lo hace más eficiente que la fuerza bruta, pero sigue siendo computacionalmente inviable para curvas elípticas de orden grande como las que se usa en la implementación.

- Pohlig-Hellman: Propuesto por Stephen Pohlig y Martin Hellman en 1978 [105], el algoritmo pretende, dado un grupo cíclico G de orden n con generador g y un elemento h en el grupo, encontrar el valor de x tal que $g^x = h$. Este ataque es efectivo si el orden del grupo al cual se pretende atacar tiene factores primos pequeños, dado que reduce el problema del logaritmo discreto a varios problemas de subgrupos de orden primario.

  Suponiendo que el orden del grupo se puede factorizar como:

$$n = p_1^{e1} p_2^{e2} p_3^{e3} \cdots p_k^{ek}$$ donde $p_i$son primos y e_i son los exponentes.

El algoritmo se podrá descomponer en problemas de subgrupos de orden p_i^{ei}, donde se resolverá el problema del logaritmo discreto.

Dado que h es el elemento a calcular, para cada primo p_i y su exponente e_i , definimos y computamos g_i tal que:

$$g_i = g^{\frac{n}{p_i^{ei}}}$$

Posteriormente se computa:

$$h_i = h^{\frac{n}{p_i^{ei}}}$$ de forma que $h_i \in g_i$

Usando la ecuación $g^x = h$ resolviendo para $g_i^{xi} = h_i$ para $x_i \in \{0, ..., p_i^{ei} - 1\}$ se obtiene el resultado:

$$x_i \ mod \ p_i^{ei}$$

Este resultado produce congruencias, por lo que se debe combinar con el teorema chino del resto [106] para obtener, finalmente:

$$\mathrm{x} \ mod \ n$$

Es importante notar que, en la implementación, se utilizan curvas elípticas cuyo orden es un número primo grande o tiene un cofactor muy pequeño, lo que hace que este ataque no sea efectivo.

- *Index Calculus* [107]: Este tipo de ataques resulta ser uno de los más efectivos a la hora de resolver el problema del logaritmo discreto en ciertos grupos, aunque no se puede aplicar directamente a las curvas elípticas que se usan. Su objetivo es, basándose en los principios de teoría de números y álgebra lineal, reducir el problema del logaritmo discreto a un sistema de ecuaciones lineales.

  Para ello primero se elige un conjunto de elementos primos pequeños denominado base $B = \{b_1, b_2, \ldots, b_m\}$ perteneciente a F_p, siendo el primer elemento el -1.

  A continuación, se procede a la generación de relaciones, es decir, se buscan elementos aleatorios en el grupo que puedan factorizarse en términos de la base B. Esto implica encontrar:

  $$g^k = \prod_{b_i \in B} b_i^{e_i} \ mod \ p$$

  donde g es el generador del grupo, mientras que e_i son los exponentes de los factores b_i.

Cada relación encontrada proporciona una ecuación en los logaritmos discretos de los elementos de la base, los cuales se emplearán para construir el sistema de ecuaciones lineales:

$$log_g(g^k) \equiv K \equiv \sum_i e_i log_g(b_i) \ mod\ (p-1)$$

Aplicando álgebra lineal para resolver el sistema de ecuaciones, se obtiene la solución al logaritmo discreto para los elementos de la base B. Una vez que se conocen los logaritmos discretos de los elementos de la base, el logaritmo discreto de cualquier elemento del grupo puede ser expresado en términos de estos logaritmos. Por tanto, para obtener el elemento h se aplica una factorización en términos de la base B, tal que:

$$h = \prod_{b_i \in B} b_i^{f_i} \ mod\ p$$

Obteniendo finalmente el logaritmo discreto de h:

$$log_g(h) \equiv \sum_i f_i log_g(b_i) \ mod\ (p-1)$$

Es importante destacar que, aunque este método es muy eficaz para ciertos grupos finitos, no se conoce una versión eficiente para curvas elípticas, lo que contribuye a la seguridad de nuestro sistema ECDH.

- El método de Rho de Pollard [108]: Es un algoritmo utilizado para resolver el problema del logaritmo discreto basado en el ciclo de los valores generados por una función pseudoaleatoria para encontrar colisiones (dos entradas diferentes que producen la misma salida) de forma eficiente. Según el autor del método, Pollard, J. M., estas coincidencias se producen aproximadamente cada 0,6267√N elementos.

  La implementación de este algoritmo tiene algunas dificultades ya que para aprovechar sus ventajas es necesario aplicar algún criterio para seleccionar qué puntos se almacenarán y cuáles no, de lo contrario se estaría en la misma situación que el algoritmo *Baby Step, Giant Step.*

  Como se ha mencionado, este algoritmo hace uso de una función pseudoaleatoria f la cual se emplea para mapear o generar puntos de la curva.
  Comenzando con el algoritmo, es necesario establecer un punto inicial:

$$P_0 \equiv a_0 P + b_0 Q$$

  donde los parámetros a_0 y b_0 son coeficientes aleatorios.
  Posteriormente, la función f se utiliza para generar una secuencia de puntos mientras se actualizan los coeficientes:

$$P_{i+1} = f(P_i)$$
$$P_i = a_i P + b_i Q$$

Una vez se empieza a producir una secuencia de valores cíclica, se aplica el algoritmo de detección de ciclos de Floyd, un método para detectar ciclos en secuencias de valores generadas por una función iterativa. El uso de este método permite detectar las colisiones de forma que:

$$P_i = P_j \text{ con } i \neq j$$

Una vez encontrada dicha colisión, se procede a resolver el problema del logaritmo discreto:

Siendo $P_i = P_j$ con $i \neq j$, entonces:

$a_i P + b_i Q = a_j P + b_j Q$, es decir $(a_i - a_j)P = (b_i - b_j)Q$

Dado que $Q = KP$, se tiene que: $(a_i - a_j)P = (b_i - b_j)KP$

Si P es de orden n, entonces se obtiene que:

$(a_i - a_j) \equiv (b_i - b_j)K \bmod n$

Y si $b_j \neq b_i$, se puede resolver K de forma que:

$K \equiv (a_i - a_j)(b_i - b_j)^{-1} \bmod n$ donde $(b_i - b_j)^{-1}$ es el inverso multiplicativo de $(b_i - b_j)$ en mod n

Una consideración importante es que este método de ataque divide n en un número d de fragmentos los cuales explora para encontrar el valor de K. Sin embargo, en las aplicaciones criptográficas es común que d sea primo, por lo tanto, d solo puede ser 1 o n. Si $d = n$, se debe elegir otro punto inicial y comenzar de nuevo, ya que los coeficientes son múltiplos de n. Sin embargo, si $d = 1$, se obtiene $K \bmod n$.

Este método tiene una complejidad de O(√n), y es similar al *Baby-Step Giant-Step*, pero con la ventaja de requerir menos memoria. Sin embargo, para las curvas elípticas que se utiliza en la implementación, con órdenes de alrededor de 2^256, este ataque sigue siendo computacionalmente inviable en la práctica [109].

- Método de Kangaroo o algoritmo lambda de Pollard [110]: Este algoritmo tiene la particularidad de ser más útil cuando el exponente que se busca se encuentra dentro de un rango determinado. λ Pollard pretende hallar el valor de x dentro del rango conocido $[a, b]$ haciendo uso de dos funciones pseudoaleatorias, una realizando '*tame kangaroo*' (canguro domesticado) y otra haciendo *'wild kangaroo*' (canguro salvaje), de ahí su nombre. El objetivo de ambas funciones

es que en algún momento las dos caigan en el mismo ciclo, permitiendo así resolver el problema del logaritmo discreto.

Para conseguir esto, el algoritmo de Pollard define dos puntos iniciales para cada función y una función de paso tal que $f: G \rightarrow S$:

$x_0 = a^b$ para *tame kangaroo*

$y_0 = \beta$ para *wild kangaroo*

El avance de cada *kangaroo* se produce de forma iterativa:

$x_{i+1} = x_i a^{f(x_i)}$ para *tame kangaroo*

$y_{j+1} = y_j a^{f(x_i)}$ para *wild kangaroo*

Sin embargo, *tame kangaroo* no puede avanzar de forma indefinida, sino que avanza un número de pasos n y luego se detiene. Pasados n saltos, se guarda la distancia recorrida por *tame kangaroo* de la siguiente forma:

$$d(n) = \sum_{i=0}^{n-1} f(y_i)$$

Mientras que *wild kangaroo* computa:

$$d = \sum_{i=0}^{n-1} f(x_i)$$

Ambas partes van avanzando hasta que se produce una colisión tal que $x_i = y_j$.

De esta forma la resolución del problema del logaritmo discreto es:

$$x_n = y_j \Rightarrow \alpha^{b+d} = \beta\alpha^{d_j} \Rightarrow \beta = \alpha^{b+d-d_j} \Rightarrow x \equiv b + d - d_j \mod n$$

Aunque, se debe tener en cuenta que el algoritmo puede fallar de forma:

$$d_j > b + d - d_j$$

El problema que tiene este algoritmo es que se desconoce cuántas iteraciones debe hacer *tame kangaroo*, por esta razón, el algoritmo es probabilístico. Sin embargo, sigue siendo una buena opción ya que solo se está buscando la solución del logaritmo discreto en un rango determinado.

En la implementación, al usar curvas elípticas con órdenes grandes y generar claves privadas de manera aleatoria en todo el rango posible, este ataque no representa una amenaza significativa.

9.3.2. Ataque de claves débiles

Los Ataques a claves débiles [111] son aquellos que tienen como objetivo claves privadas cuya estructura o valor es más susceptible de ser adivinada que una clave fuerte. La generación de estas claves débiles viene dada por el método encargado de la generación aleatoria de la clave o porque los subgrupos de puntos de la curva son pequeños. En este caso, el atacante podría usar métodos como Pohlig-Hellman para reducir la complejidad del problema del logaritmo discreto.

Ante este tipo de ataques, lo más conveniente es asegurar de la aleatoriedad del método encargado de generar la clave privada d [112] [113]. En la implementación, se utilizarán generadores de números aleatorios criptográficamente seguros para generar las claves privadas, lo que mitiga significativamente este riesgo.
Además, las dos curvas que se han elegido tienen órdenes que son números primos grandes o tienen cofactores muy pequeños, lo que elimina la posibilidad de ataques basados en subgrupos pequeños.

9.3.3. Ataque en protocolos, *Man in the middle* (MitM)

Como se ha mencionado anteriormente, el ataque *man in the middle* puede comprometer la integridad de la comunicación entre A y B si no se implementan medidas de autentificación y verificación. Aquí se abordarán las mitigaciones al ataque:

- Autenticación mutua: Los usuarios involucrados en la comunicación deben verificar la identidad del otro usuario mediante el uso de certificados digitales emitidos por una autoridad certificadora (CA) y una infraestructura de clave pública. Antes de que se establezca la comunicación segura, cada usuario verifica el certificado digital del otro usuario para evitar suplantación. De esta forma, el atacante no podrá interponerse entre la comunicación de dos usuarios impidiendo así el ataque MitM.

- Firmas Digitales: En esta técnica se busca verificar la autenticidad e integridad del mensaje haciendo uso de las claves públicas y privadas, para generar la firma y para verificarla. Aunque no se ha descrito anteriormente, ni se ha usado en el desarrollo, se debe crear el *hash* del mensaje a enviar usando una función *hash.* El usuario A deberá encriptar el *hash* obtenido con su clave privada, generando de esta forma la firma digital y enviando así el mensaje original junto con la firma. El usuario B también genera su propio *hash* y recibe

el mensaje con la firma digital de A. B desencripta la firma digital usando la clave pública de A para obtener el *hash* original y comprueba si coincide con el que ha generado. Si la firma es válida entonces el mensaje no ha sido alterado y se verifica que realmente el mensaje enviado por el usuario A.

- Validación de puntos. Esta técnica, mencionada anteriormente, es una mitigación crítica para los ataques a curvas elípticas. Si un punto no se valida correctamente para verificar que el punto pertenece a la curva un atacante podría introducir un punto invalido provocando comportamientos inesperados en el sistema comprometiendo la seguridad del mismo. Los puntos que pertenecen a subgrupos pequeños pueden ser usados como vulnerabilidad, ya que si un punto resulta tener un orden pequeño, este puede revelar parcialmente la clave privada del usuario.

  Para mitigar este riesgo, es necesario verificar que se satisface la ecuación de la curva que se emplea. También es necesario asegurar que los puntos tienen el orden correcto, es decir, el punto pertenece al subgrupo del orden primo de la curva.

  Sea $P = (x, y) \Longrightarrow nP = O$, donde n el orden del grupo y O es el punto en el infinito.

  Por último, es necesario asegurar que los puntos no pertenecen al subgrupo de torsión, un punto que multiplicado por un entero produce el punto en el infinito. Esto involucra la implementación de técnicas como proyección en el subgrupo de orden primo o el uso de curvas con un cofactor pequeño, así el número de puntos de puntos no deseable se reduce al mínimo.

En este desarrollo, se ha puesto especial esmero en validar los puntos para la curva secp256r1, como se vió en las pruebas. Para la Curve25519, la validación de puntos no es necesaria debido a su diseño, lo que simplifica la implementación y reduce el riesgo de errores en este proceso.

9.3 Ataques históricos relevantes

En esta sección se detallan brevemente ataques históricos relevantes que se estima aportan información valiosa al estudio.

9.3.1 Ataque de Lentra contra ECDSA

El ataque de Lenstra contra ECDSA (*Elliptic Curve Digital Signature Algorithm,* Algoritmo de Firma Digital de Curva Elíptica) [114] es un ataque teórico propuesto por Arjen Lentra en 2004. Este ataque se basa en la posibilidad de generar curvas elípticas débiles que podrían comprometer la seguridad del sistema si se utilizan para ECDSA.

El ataque se centra en la generación de curvas elípticas con unas propiedades especiales. Estas propiedades hacen que el problema del logaritmo discreto sea más fácil de resolver. En concreto, Lenstra propuso un método para construir curvas elípticas donde el número de puntos en la curva es un producto de factores primos pequeños.

Si un atacante pudiera convencer a un usuario de utilizar una de estas curvas débiles, podría potencialmente romper el sistema ECDSA basado en esa curva. Sin embargo, es importante notar que este ataque no afecta a las curvas estándar bien establecidas como las que se tratan en el proyecto.

La lección principal de este ataque es la importancia de utilizar curvas elípticas bien estudiadas y estandarizadas, y no generar curvas propias a menos que se tenga un profundo conocimiento criptográfico y se realicen extensas pruebas de seguridad.

9.3.2 Ataque de Dual_EC_DRBG

El ataque de Dual_EC_DRBG (*Dual Elliptic Curve Deterministic Random Bit Generator*, Generador Determinístico de Bits Aleatorios con Curva Elíptica Dual) [115] es un caso notorio en la historia de la criptografía que resalta la importancia de la transparencia en el diseño de algoritmos criptográficos.

Dual_EC_DRBG era un generador de números aleatorios basado en curvas elípticas. Fue estandarizado por NIST en 2006. Sin embargo, en 2013 se reveló que este algoritmo contenía una puerta trasera que permitía a quien conociera ciertos parámetros secretos predecir sus salidas.

El problema surgió en la elección de ciertos puntos en la curva elíptica utilizada por el algoritmo. Estos puntos, P y Q, estaban relacionados de una manera que permitía a quien conociera la relación entre ellos predecir las salidas del generador.

Este incidente causó una gran controversia en la comunidad criptográfica y llevó a una mayor examinación de los estándares criptográficos. Esto explicaría la inseguridad alrededor de secp256r1. También resaltó la importancia de utilizar generadores de números aleatorios bien estudiados y de confiar en la aleatoriedad del sistema operativo cuando sea posible. En esta implementación, se ha tenido cuidado de utilizar fuentes de aleatoriedad confiables y bien estudiadas.

9.4 Ataques de canal lateral

Los ataques de canal lateral representan una amenaza significativa y creciente en la criptografía moderna. Según un estudio reciente de la Universidad de Maryland, aproximadamente el 15% de los ataques criptográficos exitosos en los últimos 5 años han sido ataques de canal lateral [116]. Estos ataques intentan obtener información a través de la información que se filtra durante la ejecución del algoritmo, como el tiempo de ejecución, el consumo de energía o incluso la emisión de pequeños campos electromagnéticos.

Los más relevantes son:

- *Timing Attack* [117] (Ataque de temporización): En estos ataques el tiempo en las operaciones criptográficas depende de los parámetros de entrada y un atacante puede observar dichos tiempos para intentar obtener la clave. En el contexto de las curvas elípticas, un atacante podría intentar medir el tiempo que toma la operación de multiplicación de puntos para deducir información sobre la clave privada.

  Para protegerse contra este tipo de ataques, es vital implementar operaciones de tiempo constante. En la implementación de ECDH, se ha tenido cuidado de utilizar algoritmos de multiplicación de puntos que realizan el mismo número de operaciones independientemente del valor de la clave privada.

- *Power Analysis Attack* (Ataque por Análisis de Fallo Diferencial) [118]: El objetivo de este ataque es medir el consumo del dispositivo hardware durante las operaciones para analizarlo y extraer la clave privada. Existen dos variantes principales:
  - *Simple Power Analysis* (SPA): El atacante intenta deducir información directamente de una sola traza de consumo de energía.
  - *Differential Power Analysis* (DPA) [119] [120]: Explicado en la siguiente sección.

Para mitigar esto, se pueden implementar técnicas de enmascaramiento y aleatorización. En una implementación hardware, como la realizada en FPGA, se pueden utilizar técnicas de diseño balanceado para que el consumo de energía sea lo más uniforme posible independientemente de los datos procesados.

- *Cache Timing Attack* (Ataque de Temporización de Caché) [121]: Este ataque se centra en observar los patrones de acceso a la memoria caché para deducir la clave privada. Es particularmente relevante en implementaciones software que utilizan tablas precalculadas, como las S-boxes en AES.

  En la implementación de AES, se ha optado por utilizar una S-box precalculada. Aunque esto puede potencialmente hacer la implementación más vulnerable a ataques de caché en un contexto de software, en la implementación en FPGA este riesgo se mitiga significativamente. En el contexto de hardware dedicado, el uso de una S-box precalculada ofrece ventajas en términos de rendimiento sin comprometer la seguridad de la manera en que lo haría en una implementación puramente software.

- EMA (*Electromagnetic Analysis*, Ataque por Análisis Electromagnético) [122]: Basado en las emisiones electromagnéticas del dispositivo durante las operaciones criptográficas. En una implementación en FPGA, se pueden utilizar técnicas de diseño que minimicen las emisiones electromagnéticas, como el uso de filtros y blindaje adecuado.

Es importante destacar que la protección completa contra todos los ataques de canal lateral es un desafío continuo en el campo de la criptografía práctica.

9.5 Consideraciones de seguridad para implementaciones en hardware

La implementación de algoritmos criptográficos en hardware, como en este caso con FPGA, presenta desafíos y consideraciones de seguridad únicos. A continuación, se explicará algunos de los aspectos más importantes:

- Ataques de implementación: Estos ataques se centran en las debilidades de cómo se ha implementado el algoritmo a nivel de hardware. Algunos de los más relevantes son:
  - *Differential Fault Analysis* (Análisis de Fallos Diferenciales) [123]*:* Este ataque intenta introducir fallos físicos en el hardware para producir errores en los resultados de las operaciones criptográficas. De esta forma, el atacante puede

deducir la clave privada comparando los resultados correctos con los erróneos. Se sugiere la inclusión futura de técnicas de detección y corrección de errores, así como redundancia en las operaciones críticas para detectar y prevenir la propagación de errores inducidos.

- *Glitch Attacks* (Ataque por Interrupciones) [124]*:* Estos ataques manipulan la señal de reloj o la alimentación del dispositivo para inducir comportamientos erróneos. Se recomienda considerar la inclusión de circuitos de monitoreo de voltaje y frecuencia en futuras versiones para detectar y prevenir este tipo de manipulaciones.

- Seguridad del *bitstream*: En las FPGAs, el *bitstream* es el archivo que contiene la configuración del hardware. Proteger este archivo es crucial para la seguridad del sistema y recomendable implementar encriptación del *bitstream* y autenticación para prevenir la lectura no autorizada o la modificación del diseño.

- Protección contra ingeniería inversa: Las FPGAs son vulnerables a ataques de ingeniería inversa que podrían revelar detalles de la implementación. Por ello, es recomendable la incorporación de técnicas como la ofuscación del diseño y la aplicación de capas adicionales de seguridad. Aunque las herramientas de desarrollo proporcionadas por las compañías dificultan este tipo de ataques.

- *Side-channel attack resistance* (Resistencia a Ataques de Canal Lateral) [125]: Como se mencionó anteriormente, los ataques de canal lateral son una preocupación importante en implementaciones hardware. Por esta razón es fundamental considerar técnicas como el balanceo de carga y la inserción de ruido para mitigar estos ataques.

- Protección contra ataques de sondeo: Los ataques de sondeo implican la inserción de sondas físicas dentro del dispositivo para extraer información crítica. Estos ataques son particularmente peligrosos porque pueden proporcionar acceso directo a las claves criptográficas y otros datos sensibles sin necesidad de descifrar el tráfico cifrado. Ante este tipo de ataques, se recomienda la utilización de materiales de protección activa que reaccionen ante intentos de sondeo, como el uso de capas conductoras que activen una alarma al ser perforadas.

- Gestión segura de claves: La gestión de claves es un aspecto crítico en cualquier sistema criptográfico, pero especialmente en implementaciones hardware. Por este motivo es imprescindible implementar técnicas de almacenamiento seguro y generación de claves robustas.

- Actualización segura: Es importante poder actualizar el firmware de manera segura para corregir vulnerabilidades o mejorar el rendimiento. Por ello, es de gran importancia que se usen procedimientos de autenticación y verificación de integridad para las actualizaciones de firmware.

Aunque no se han implementado mitigaciones específicas para estos tipos de ataques, estas recomendaciones servirán de guía para futuras mejoras de seguridad en la implementación hardware del criptosistema.

Es importante destacar que, aunque estas medidas proporcionan un alto nivel de seguridad, la protección absoluta contra todos los ataques posibles es prácticamente imposible. Por lo tanto, es crucial mantener un enfoque de seguridad en capas, combinando estas protecciones a nivel de hardware con buenas prácticas de seguridad a nivel de sistema y aplicación.

10. Aspectos prácticos y aplicaciones reales

La implementación pura de un criptosistema basado en ECDH para el intercambio de claves y AES-128 en modo CTR para el cifrado que se ha desarrollado tiene numerosas aplicaciones potenciales en el mundo real. Es importante señalar que, antes de su uso en entornos de producción, debería ser sometido a rigurosas pruebas de seguridad. Sin embargo, el trabajo puede ser utilizado en simulaciones de laboratorio y estudios de seguridad.

A continuación, se exploran algunas de las aplicaciones prácticas más relevantes.

- Seguridad de comunicaciones en redes:
    - VPNs (*Virtual Private Network*, Redes Privadas Virtuales): ECDH se puede utilizar para el establecimiento de una comunicación segura entre el cliente y el servidor VPN. AES puede cifrar los datos transmitidos, protegiendo la privacidad y la integridad de la información.
    - Protocolos de Seguridad en Internet (TLS/SSL): En la configuración de conexiones HTTPS, ECDH se usa para el intercambio seguro de claves durante el *handshake* (negociación para establecer las condiciones de la comunicación), y AES puede ser utilizado para cifrar la sesión de datos.
- Dispositivos IoT: Aunque los dispositivos IoT normalmente no tienen los recursos para implementar FPGAs, los *gateways* que manejan la comunicación de múltiples dispositivos podrían beneficiarse de ellas.
- Comunicación Segura entre Dispositivos: Los dispositivos IoT pueden utilizar ECDH para el intercambio de claves y AES para cifrar los datos enviados.
- Actualizaciones de Firmware Seguras: Para actualizar el firmware de dispositivos IoT, ECDH puede asegurar que solo dispositivos autenticados puedan recibir y aplicar actualizaciones, mientras que AES puede cifrar el firmware durante la transmisión para protegerlo contra manipulaciones.
- Seguridad en *Blockchain*: En operaciones de minería de criptodivisas, la FPGA podría utilizarse para asegurar la comunicación entre nodos de minería, protegiendo contra ataques de intermediarios.
- Carteras digitales: Uso en las carteras digitales para proteger las transacciones y la información de la cartera para proteger contra robos y manipulaciones.

- Contratos inteligentes: Uso en las plataformas de contratos inteligentes asegurando que las condiciones y los resultados del contrato sean confidenciales y no alterables.
- Aceleración de operaciones criptográficas en centros de datos: La FPGA podría funcionar como un acelerador criptográfico dedicado en centros de datos, manejando el intercambio de claves ECDH y el cifrado AES para múltiples conexiones en paralelo.
- Seguridad en redes de alto rendimiento: La implementación FPGA es particularmente útil en escenarios de red que requieren un alto rendimiento de datos cifrados.
- Sistemas de control industrial: En entornos industriales donde la seguridad y el rendimiento son esenciales, esta implementación FPGA podría proporcionar cifrado de alta velocidad y baja latencia.
- Comunicaciones militares y gubernamentales seguras: La naturaleza del desarrollo hardware realizado FPGA la podría hacer, con implementación de algoritmos más avanzados, adecuada para aplicaciones de seguridad nacional que requieren el más alto nivel de seguridad y rendimiento.

11. Impacto social y medioambiental

Hoy en día, el acceso a Internet ha pasado a ser una rutina y es por ello que se debe tener más constancia de la huella digital y los datos personales que se facilitan. La creciente preocupación por este tema es el que ha inspirado el auge de muchos algoritmos que se pueden encontrar hoy en día, cuyos objetivos son asegurar que los datos sensibles de los usuarios no sean interceptados y manipulados por terceros malintencionados.

Actualmente es imprescindible que en una comunicación a través de Internet se pueda verificar la identidad de los usuarios y la integridad de los datos para evitar estas brechas de riesgos que puedan exponer la información confidencial. Al final, la confianza en los sistemas criptográficos es fundamental para que estos lleguen a tener un uso generalizado, como se ha visto con el caso de las curvas usadas en varios servicios a día de hoy. La implementación de algoritmos conocidos, la documentación clara de estos y su transparencia es la que permite a expertos y usuarios verificar la seguridad del sistema, generando la confianza de que los datos han sido protegidos y que el sistema funciona correctamente de forma predecible y verificable [126].

Aunque esto también implica que el diseño del sistema o algoritmo debe ser prevenido de un mal uso como el uso indebido en actividades delictivas. Por ello se incluyen controles de acceso, monitorización y aplicaciones de leyes y regulaciones que afectan a las empresas y cómo implementan estas tecnologías. En este sentido, este trabajo en criptografía contribuye a la creación de sistemas más seguros, pero también plantea cuestiones éticas sobre el equilibrio entre privacidad y seguridad pública.

No solo se debe tener constancia de la huella digital, también se debe considerar la huella de carbono que se deja. El uso de FPGAs en este ámbito de criptología resulta ser más eficiente energéticamente que las CPUs y GPUs. Este es un aspecto fundamental para reducir la huella de carbono generada en el planeta y por esta razón cada vez busca implementar algoritmos que maximicen la eficiencia energética, contribuyendo a la sostenibilidad ambiental y la reducción de los costes operativos.

Las implementaciones eficientes buscan minimizar los recursos usados incluyendo el diseño de arquitecturas que aprovechen al máximo las capacidades del hardware, reduciendo la necesidad de ciclos de procesamiento adicionales y minimizando la redundancia. En este

caso, la implementación de AES y ECDH en FPGA no solo mejora el rendimiento, sino que también contribuye a la eficiencia energética global de los sistemas de seguridad.

Además, como se mencionó al principio, las FPGAs tienen un ciclo de vida prolongado ya que se pueden reprogramar y ser empleadas en diferentes aplicaciones. Esta característica reduce la necesidad de estar fabricando nuevo hardware o dispositivos reduciendo la generación de residuos tecnológicos. Esto se alinea con los principios de la economía circular, promoviendo la reutilización y adaptación de hardware existente en lugar de su reemplazo constante.

Es importante también destacar el impacto social de la criptografía en la protección de derechos humanos y libertades civiles. Este trabajo en algoritmos de encriptación robustos puede contribuir a proteger la privacidad de activistas, periodistas y ciudadanos en regímenes represivos, facilitando la libre expresión y el intercambio seguro de información.

Por último, el desarrollo de sistemas criptográficos eficientes y seguros tiene un impacto directo en la inclusión digital. Al hacer que las comunicaciones seguras sean mucho más accesibles y eficientes, se contribuye a que más personas puedan participar de manera segura en la era digital, independiente de sus recursos tecnológicos.

12. Conclusión

En este proyecto se ha tratado en profundidad el criptosistema conformado por dos de los algoritmos más usados y seguros en la actualidad: AES-128 y ECDH. Asimismo, se ha abordado la implementación de dicho criptosistema sobre un dispositivo hardware programable, la FPGA.

Para alcanzar estos objetivos, el proyecto se ha dividido en tres etapas: documentación y análisis de los algoritmos, implementación software e implementación hardware. La implementación se ha dividido en dos partes para facilitar el desarrollo del hardware y asegurar un código bien estructurado.

La investigación y el análisis del algoritmo AES (en sus modos CTR, ECB y CBC) junto con Diffie-Hellman de curvas elípticas (específicamente Curve25519 y secp256r1) ha permitido comprender en profundidad el funcionamiento de cada uno, así como sus ventajas y desventajas, y las razones de su amplio uso en la actualidad. Como parte del análisis, también se ha explorado la variedad de métodos de ataque a estos algoritmos, lo que ha enriquecido la comprensión de sus vulnerabilidades y fortalezas.

La implementación en hardware ha presentado ciertos desafíos, especialmente en lo que respecta al uso del protocolo UART para las comunicaciones con dispositivos externos. Esto ha requerido el desarrollo de máquinas de estado para controlar el envío y recepción de datos a través de dicho protocolo. Aunque no se logró una implementación completamente funcional en el hardware debido a errores persistentes, sí se pudo realizar una simulación correcta del criptosistema, obteniendo datos interesantes sobre su rendimiento.

El análisis detallado de los diversos ataques y consideraciones de seguridad, tanto para AES y ECDH como para la implementación en hardware, ha llevado a varias conclusiones importantes:

- Seguridad en capas: Este trabajo demuestra la importancia de un enfoque de seguridad multinivel, desde la elección de algoritmos robustos hasta las consideraciones específicas de la implementación en hardware.

- Limitaciones de la seguridad: Se reconoce que no existe un sistema completamente invulnerable, lo que subraya la necesidad de un enfoque proactivo y en constante evolución.
- Valor de los estándares: La elección de algoritmos estandarizados como AES y curvas elípticas bien establecidas ha sido fundamental para garantizar un nivel base de seguridad.
- Equilibrio entre eficiencia y seguridad: Se ha enfrentado a decisiones que implican un compromiso entre estos dos aspectos, buscando siempre priorizar la seguridad sin comprometer excesivamente el rendimiento.
- Importancia de la implementación: Se ha constatado que incluso con algoritmos teóricamente seguros, una implementación deficiente puede introducir vulnerabilidades significativas.
- Desafíos del hardware: Esta experiencia con la FPGA ha demostrado que, si bien el hardware puede ofrecer ventajas en términos de seguridad, también introduce nuevas consideraciones y desafíos técnicos.

En resumen, este proyecto ha permitido profundizar en la complejidad de los sistemas criptográficos modernos, desde su base teórica hasta los desafíos prácticos de su implementación en hardware. Aunque no se ha logrado una implementación completamente funcional en la FPGA, las lecciones aprendidas y los resultados de las simulaciones proporcionan una base sólida para futuros desarrollos y mejoras en este campo.

12.1. Líneas futuras

Para futuras versiones del proyecto, se pueden contemplar cambios y nuevas funcionalidades que se podrían añadir:

- Optimización de la implementación en FPGA: Resolver los errores que impidieron la implementación completa en hardware es una prioridad. Esto permitiría realizar pruebas de rendimiento reales y comparar con los resultados de la simulación.
- Exploración de otras curvas elípticas: Aunque el trabajo se ha centrado en Curve25519 y secp256r1, sería valioso implementar y comparar el rendimiento de otras curvas elípticas populares.

- Exploración de otros modos AES: Del mismo modo, sería recomendable implementar modos autenticados que mejorarían mucho la seguridad, en particular GMC.
- Implementación de contramedidas contra ataques de canal lateral: Desarrollar y evaluar técnicas específicas para proteger la implementación en FPGA contra ataques de este estilo.
- Mejorar el generador de números aleatorios: Investigar e implementar TRNG (*True Random Number Generato*r, Generador de Números Aleatorios Verdaderos) [127] basado en hardware para mejorar la seguridad del sistema.
- Exploración de criptografía post-cuántica: Dado el avance en este campo, sería interesante explorar la implementación de estos en FPGAs.
- Optimización: Buscar optimizaciones tanto en el diseño lógico como en la utilización de los recursos de la FPGA [128].
- Análisis de seguridad ampliado: Realizar un análisis de seguridad más exhaustivo, incluyendo pruebas de penetración y evaluación de vulnerabilidades específicas de la implementación en la FPGA.

13. Referencias

[1] GeeksforGeeks. (n.d.). Advanced Encryption Standard (AES). GeeksforGeeks https://www.geeksforgeeks.org/advanced-encryption-standard-aes/.

[2] Wikipedia contributors. (n.d.). Diffie-Hellman key exchange. https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange

[3] Cabrera Aldaya, A., & Cabrera Sarmiento, A. J. (2013). Design and Integration of Cryptographic Algorithms on FPGA-Based Embedded Systems.34(3).

[4] Wikipedia contributors. (n.d.). Field-programmable gate array. https://en.wikipedia.org/wiki/Field-programmable_gate_array

[5] Wikipedia contributors. (n.d.). Elliptic-curveDiffie-Hellman https://en.wikipedia.org/wiki/Montgomery_curve

[6] NIST, "Post-Quantum Cryptography: NIST's Plan for the Post-Quantum World," National Institute of Standards and Technology, U.S. Department of Commerce, 2020. [Online]. https://www.nist.gov/post-quantum-cryptography

[7] D. J. Bernstein, J. Buchmann, E. Dahmen. (2009). Post-Quantum Cryptography: Lattice-based and Hash-based Cryptosystems. Springer-Verlag Berlin Heidelberg.

[8] J. Ding, J. E. Gower, D. S. Schmidt. (2020). Multivariate Public Key Cryptosystems. Advances in Information Security, Springer.

[9] Netmeister. (2024, January 1). Post-quantum crypto: 2024 and beyond. Netmeister. https://www.netmeister.org/blog/pqc-2024-01.html

[10] IBM Research Blog. "How Homomorphic Encryption is Changing Cloud Security." IBM Security Blog, 2021. https://www.ibm.com/blogs/research

[11] S. Goldwasser, Y. Kalai, G. N. Rothblum. (2023). Zero-Knowledge Proofs: From Theory to Practice in Blockchain Systems. Communications of the ACM, Vol. 66 No. 5.

[12] Wikipedia contributors. (n.d.). Edwards curve. In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Edwards_curve

[13] Wikipedia contributors. (n.d.). Montgomery curve. https://en.wikipedia.org/wiki/Montgomery_curve

[14] D. J. Bernstein, T. Lange. (2020). SafeCurves: choosing safe curves for elliptic-curve cryptography. https://safecurves.cr.yp.to

[15] Ars Technica. (2022, August 23). SIKE, once a post-quantum encryption contender, is KO'd in NIST smackdown. Ars Technica.

[16] M. Marlinspike, T. Perrin. (2016). The Double Ratchet Algorithm. Signal Foundation

[17] WhatsApp. (2023). WhatsApp Encryption Overview: Technical white paper.

[18] D. Hankerson, A. Menezes, S. Vanstone. (2022). Guide to Elliptic Curve Cryptography. Springer Science & Business Media.

[19] Wang, Y., & Shankar, U. (2018). An overview of side-channel attacks and defences. University of Maryland.

[20] Apple. (n.d.). Introducción a la seguridad de macOS. https://support.apple.com/es-es/guide/security/sec59b0b31ff/web

[21] Intel Corporation. (2023). Intel® Software Guard Extensions (Intel® SGX). Intel Developer Documentation.

[22] ARM Limited. (2023). ARM TrustZone Technology Overview. ARM Developer Documentation.

[23] Microsoft. (n.d.). Trusted Platform Module (TPM) overview. https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/trusted-platform-module-overview

[24] Cloudflare. (2024). Post-Quantum Cryptography 2024: NIST's Selections. Cloudflare. https://blog.cloudflare.com/pq-2024

[25] Boston Consulting Group. (2024). Long-term forecast for quantum computing still looks bright.https://www.bcg.com/publications/2024/long-term-forecast-for-quantum-computing-still-looks-bright

[26] Wikipedia contributors. (n.d.). 51% attack. https://en.wikipedia.org/wiki/51%_attack

[27] Chainalysis. (2024). Crypto hacking and stolen funds: What we learned in 2024. https://www.chainalysis.com/blog/crypto-hacking-stolen-funds-2024/

[28] Schapeler, E. T., Starnecker, N., Weiser, C., & Florian. (2021). Efficient FPGA Implementation of Blockchain Operations. Chair of Computer Architecture and Parallel Systems, Technical University of Munich.

[29] Lahiri, S., Chatterjee, K. M., & Mitra, S. K. (2017). A New Crypto-based Processor System Design Using FPGA. AMSE JOURNALS-AMSE IIETA publication-2017-Series: Advances D; Vol. 22; N°1; pp 122-136.

[30] Nikhilesh Nayak, Akshay Chandak, Nisarg Shah y B Karthikeyan. (2017). Encryption and decryption using FPGA.

[31] Samanta, S. (2008). FPGA Implementation of AES Encryption and Decryption. Sardar Vallabhbhai National Institute of Technology, Sura.

[32] Pöppelmann, T., & Güneysu, T. (2019). High-Speed and Area-Efficient FPGA Implementations of AES-GCM for 100G Ethernet

[33] Moradi, A., et al. (2011). Lightweight and Fault-resistant FPGA Implementation of the Advanced Encryption Standard.

[34] Güneysu, T., & Moradi, A. (2011). Side-Channel Attack Resistant FPGA Implementation of AES.

[35] Toubal, A., Bengherbia, B., Ould Zmirli, M., & Guessoum, A. (2020). FPGA implementation of a wireless sensor node with built-in security coprocessors for secured key exchange and data transfer. Measurement, 107429.

[36] Xu, L., Chen, L., Gao, Z., Kim, H., Suh, T., & Shi, W. (2020). FPGA based Blockchain System for Industrial IoT. 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 876-8831.

[37] Imaña, J. L., & Luengo, I. (2020). FPGA Implementation of Post-Quantum DME Cryptosystem. 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).

[38] Zeraoulia, K., & Lallali, M. (n.d). FPGA-based implementation of elliptic curve Diffie-Hellman key exchange protocol.

[39] Javeed, K., & Wang, X. (2016). Efficient FPGA Implementation of Elliptic Curve Cryptography over Prime Fields. In 2016 International Conference on Field-Programmable Technology (FPT) (pp. 185-192). IEEE.

[40] Ghosh, S., Roychowdhury, D., & Das, A. (n.d). Efficient FPGA implementation of elliptic curve cryptography and SHA-3 algorithms.

[41] Celi Méndez, J. A., & Ponguillo Intriago, R. A. (2013). Implementación Hardware del Estandar de Encriptación Avanzado (AES) en una FPGA. Facultad de Ingeniería en national Electricidad y Computación, Escuela Superior Politécnica del Litoral.

[42] Wikipedia contributors. (n.d.). Quantitative research. https://en.wikipedia.org/wiki/Quantitative_research

[43] Katz, J., & Lindell, Y. (2007). Introduction to modern cryptography: Principles and protocols. Chapman & Hall/CRC.

[44] Visual Studio Code. (n.d.). Visual Studio Code - Code Editing. Redefined https://code.visualstudio.com/

[45] Python Software Foundation. (n.d.). PEP 8 -- Style guide for Python code. Python Enhancement Proposals.https://peps.python.org/pep-0008/

[46] Python Steering Council. (n.d.). PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/

[47] Xilinx Inc. (n.d.). Vivado Design Suite - HLx Editions. https://www.xilinx.com/products/design-tools/vivado.html

[48] RISC-V International. (2024). RISC-V: The Free and Open RISC Instruction Set Architecture. https://riscv.org/

[49] Schneier, B. (2015). Applied cryptography: protocols, algorithms, and source code in C. John Wiley & Sons.

[50] Nolting, S. (2024). NEORV32: A small, customizable and extensible MCU-class 32-bit RISC-V soft-core CPU and microcontroller-like SoC written in platform-independent VHDL. https://github.com/stnolting/neorv32

[51] Siemens Digital Industries Software. (2024). Catapult C++/SystemC High-Level Synthesis Tool. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/

[52] Wikipedia contributors. (n.d.). *Throughput* (computing). https://en.wikipedia.org/wiki/Throughput_(computing)

[53] PyCryptodome. (n.d.). Welcome to PyCryptodome's documentation. https://pycryptodome.readthedocs.io/en/latest/

[54] L. E. Bassham III, et al. (2010). A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. NIST Special Publication 800-22 Revision 1a. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf

[55] Wikipedia contributors. (n.d.). SIMD (Single Instruction Multiple Data). https://en.wikipedia.org/wiki/SIMD

[56] Digilent Inc. (n.d.). Nexys A7: FPGA Trainer Board Recommended for ECE Curriculum. https://digilent.com/reference/programmable-logic/nexys-a7/start

[57] National Institute of Standards and Technology. (2001). Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197.

[58] Wikipedia contributors. (n.d.). Data Encryption Standard. https://en.wikipedia.org/wiki/Data_Encryption_Standard

[59] Wikipedia contributors. (n.d.). Advanced Encryption Standard. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard#History

[60] Dworkin, M. J. (2001). Recommendation for Block Cipher Modes of Operation: Methods and Techniques. National Institute of Standards and Technology (NIST) Special Publication 800-38A.

[61] Lenstra, A. K., & Verheul, E. R. (2001). Selecting cryptographic key sizes. Journal of Cryptology, 14(4), 255-293.

[62] Wikipedia contributors. (n.d.). Feistel cipher. https://en.wikipedia.org/wiki/Feistel_cipher

[63] Intel Corporation. (n.d.). Intel® Advanced Encryption Standard New Instructions (Intel® AES-NI). https://www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard-aes/data-protection-aes-general-technology.html

[64] NIST. (2001). Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Special Publication 800-38A. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[65] Wikipedia contributors. (n.d.). Transport Layer Security. https://en.wikipedia.org/wiki/Transport_Layer_Security

[66] Wikipedia contributors. (n.d.). IPsec. https://en.wikipedia.org/wiki/IPsec

[67] Wikipedia contributors. (n.d.). Block cipher mode of operation - Cipher Feedback (CFB). https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_Feedback_(CFB)

[68] Wikipedia contributors. (n.d.). Block cipher mode of operation - Output Feedback (OFB). https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Output_Feedback_(OFB)

[69] NIST. (2007). Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Special Publication 800-38D. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

[70] Wikipedia contributors. (n.d.). CCM mode. https://en.wikipedia.org/wiki/CCM_mode

[71] IETF. (n.d.). PKCS #7: Cryptographic Message Syntax Version 1.5. https://datatracker.ietf.org/doc/html/rfc2315

[72] Wikipedia contributors. (n.d.). Endianness. https://en.wikipedia.org/wiki/Endianness

[73] National Institute of Standards and Technology (NIST). (n.d.). Advanced Encryption Standard Algorithm Validation Suite (AESAVS).

[74] Python Software Foundation. (n.d.). secrets — Generate secure random numbers for managing secrets. https://docs.python.org/3/library/secrets.html

[75] Schneier, B. (2015). Cryptography engineering: Design principles and practical applications. John Wiley & Sons.

[76] Ferguson, N., Schneier, B., & Kohno, T. (2010). Cryptography Engineering: Design Principles and Practical Applications. John Wiley & Sons.

[77] Hellekalek, P. (1998). Good random number generators are (not so) easy to find. Mathematics and Computers in Simulation, 46(5-6), 485-505.

[78] Rogaway, P. (2011). Evaluation of some blockcipher modes of operation. Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan.

[79] Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE Transactions on Information Theory, 22(6), 644-654.

[80] Gallagher, P. D. (2013). Digital Signature Standard (DSS). Federal Inf. Process. Stds.

[81] Gueron, S., & Krasnov, V. (2013). Fast prime field elliptic curve cryptography with 256 bit primes. Journal of Cryptographic Engineering, 3(3), 187-201.

[82] Wikipedia contributors. (n.d.). Curve448. https://en.wikipedia.org/wiki/Curve448

[83] Adalier, M. (2013). Efficient and Secure Elliptic Curve Cryptography Implementation of Curve P-256. Proceedings of the International Conference on Information Technology: New Generations (ITNG 2013), 1056-1058.

[84] Wikipedia contributors. (n.d.). Curve25519. https://en.wikipedia.org/wiki/Curve25519

[85] Wikipedia contributors. (n.d.). EdDSA. https://en.wikipedia.org/wiki/EdDSA

[86] Wikipedia contributors. (n.d.). Menezes–Qu–Vanstone protocol. https://en.wikipedia.org/wiki/Menezes–Qu–Vanstone_protocol

[87] Wikipedia contributors. (n.d.). Secp192k1. https://en.wikipedia.org/wiki/Secp192k1

[88] Brown, D. R. L. (2010). SEC 2: Recommended Elliptic Curve Domain Parameters.

[89] Bernstein, D. J. (2006). Curve25519: New Diffie-Hellman speed records. Proceedings of the 9th International Conference on Theory and Practice in Public-Key Cryptography (PKC 2006), 207-228.

[90]

[91] Soto, J. (1999). Statistical testing of random number generators. In Proceedings of the 22nd National Information Systems Security Conference (Vol. 10, No. 99, pp. 12-16). NIST.

[92] Wikipedia contributors. (n.d.). National Security Agency. https://en.wikipedia.org/wiki/National_Security_Agency

[93] Schneier, B. (2007, November 15). Did NSA put a secret backdoor in new encryption standard? Wired. https://www.wired.com/2007/11/nsa/

[94] Wikipedia contributors. (n.d.). Key derivation function. https://en.wikipedia.org/wiki/Key_derivation_function

[95] Digilent Inc. (n.d.). Basys 3 FPGA Board Reference Manual. https://digilent.com/reference/programmable-logic/basys-3/start

[96] Digilent Inc. (n.d.). Zybo Z7: Zynq-7000 ARM/FPGA SoC Development Board. https://digilent.com/reference/programmable-logic/zybo-z7/start

[97] Digilent Inc. (n.d.). Nexys A7: FPGA Trainer Board. https://digilent.com/reference/programmable-logic/nexys-a7/start

[98] Wikipedia contributors. (n.d.). Universal asynchronous receiver-transmitter. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

[99] DigiKey TechForum (2021 mar). UART (VHDL) - Semiconductor / Logic Design - Electronic Component and Engineering Solution Forum - TechForum | DigiKey. https://forum.digikey.com/t/uart-vhdl/12670

[100] R. C. Martin. (2000). Design Principles and Design Patterns. Object Mentor. https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[101] D. E. Knuth. (1997). The Art of Computer Programming, Volume 2: Seminumerical Algorithms. Addison-Wesley Professional, 3rd Edition.

[102] Stack Overflow. (n.d.). VHDL Random Number Generator. https://stackoverflow.com/questions/tagged/vhdl+random

[103] Iwan M. Duursma, Pierrick Gaudry, François Morain. (2010 Aug). Speeding up the Discrete Log Computation on Curves with Automorphisms.

[104] D. Shanks. (1971). Class Number, a Theory of Factorization and Genera. Proceedings of Symposia in Pure Mathematics, Vol. 20, American Mathematical Society.

[105] S. Pohlig, M. Hellman. (1978). An Improved Algorithm for Computing Logarithms over GF(p) and Its Cryptographic Significance. IEEE Transactions on Information Theory, Vol. 24, No. 1.

[106] Wikipedia contributors. (n.d.). Chinese remainder theorem. https://en.wikipedia.org/wiki/Chinese_remainder_theorem

[107] A. Joux. (2009). Algorithmic Cryptanalysis. Chapman & Hall/CRC Cryptography and Network Security Series.

[108] Edlyn Teske. (2001). On random walks for Pollard's rho method, 809–825.

[109] Montgomery, P. L. (1987). Speeding the Pollard and Elliptic Curve Methods of Factorization. Mathematics of Computation, 48(177), 243-264.

[110] J. M. Pollard. (1978). Monte Carlo Methods for Index Computation mod p. Mathematics of Computation, Vol. 32, No. 143, pp. 918-924.

[111] Lenstra, A. K. (2004). Key Generation Weakness in the ECDSA Standard. Technical Report, Citibank.

[112] Pardhu, T., Naga Teja, T., Bhushan, K. N., & Rani, N. U. (2014). Generation of cryptographically secured pseudo random numbers using FPGA. International Journal of Electronics and Communication Engineering & Technology (IJECET), 5(2), 21-29.

[113] Gil Marco, D. (2020). Generador hardware de claves basado en funciones físicamente no clonables. Universidad de Zaragoza

[114] A. K. Lenstra. (2004). The Security of ECDSA. RSA Laboratories Technical Note.

[115] NIST. (2014). Recommendation for the Transition from the Use of Dual_EC_DRBG. https://csrc.nist.gov/news/2014/the-transition-from-the-use-of-dual_ec_drbg

[116] D. Genkin, et al. (2023). A Survey of Recent Side-Channel Attacks on Cryptographic Implementations. Journal of Cryptographic Engineering, Springer.

[117] Kocher, P. C. (1996). Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Annual International Cryptology Conference (pp. 104-113). Springer, Berlin, Heidelberg.

[118] Hertz, J. (2023). An Introduction to Power Analysis Side Channel Attacks.

[119] Kocher, P., Jaffe, J., & Jun, B. (1999). Differential Power Analysis. En Advances in Cryptology. (pp. 388-397).

[120] Kocher, P., Jaffe, J., Jun, B., & Rohatgi, P. (2011). Introduction to Differential Power Analysis. Journal of Cryptographic Engineering, 1, 5-27.

[121] Wikipedia contributors. (n.d.). Cache timing attack. https://en.wikipedia.org/wiki/Cache_timing_attack

[122] P. Kocher, J. Jaffe, B. Jun. (1999). Differential Power Analysis. Advances in Cryptology (CRYPTO '99), Springer.

[123] E. Biham, A. Shamir. (1997). Differential Fault Analysis of Secret Key Cryptosystems. Advances in Cryptology (CRYPTO '97), Springer.

[124] O. Kömmerling, M. G. Kuhn. (1999). Design Principles for Tamper-Resistant Smartcard Processors. USENIX Workshop on Smartcard Technology.

[125] Wikipedia contributors. (n.d.). Side-channel attack. https://en.wikipedia.org/wiki/Side-channel_attack

[126] Rogaway, P. (2011). The perverse consequences of the distinction between ‘‘basic’’ and ‘‘applied’’ research. Cryptology ePrint Archive, Report 2011/564. https://eprint.iacr.org/2011/564

[127] W. Schindler. (2009). Random Number Generators for Cryptographic Applications. Cryptographic Engineering, Springer.

[128] Moradi, A., Poschmann, A., Ling, S., Paar, C., & Wang, H. (2011). Pushing the limits: A very compact and a threshold implementation of AES. In Annual International Conference on the Theory and Applications of Cryptographic Techniques (pp. 69-88). Springer, Berlin, Heidelberg.

APÉNDICES

Apéndice A

Implementación práctica del algoritmo AES

Este apéndice ilustra el proceso de cifrado completo del algoritmo AES-128 mediante un ejemplo detallado, desde los datos de entrada hasta el resultado cifrado, mostrando cada paso intermedio de las operaciones que componen las rondas del algoritmo.

A.1 Descripción del ejemplo

Este ejemplo utiliza el algoritmo AES-128 para cifrar el mensaje hexadecimal `"00112233445566778899aabbccddeeff"` con la clave hexadecimal `"000102030405060708090a0b0c0d0e0f"`. Cada paso muestra los datos transformados en cada fase del proceso de cifrado AES, incluyendo la expansión de clave y las 10 rondas del algoritmo.

A.2 Expansión de Clave

En AES-128, la clave inicial de 128 bits se expande para generar 11 subclaves. Aquí se muestran las claves generadas para las rondas iniciales y finales:

1. Clave Original: `000102030405060708090a0b0c0d0e0f`
2. Proceso de Expansión de Clave:En cada ronda, se realiza una transformación en el último grupo de bytes de la clave anterior.
    - RotWord: Se rota el último grupo de bytes.
    - SubWord: A cada byte del resultado rotado se le aplica la sustitución S-box.
    - Rcon: El resultado se XOR con una constante específica (Rcon) que varía en cada ronda.
3. Subclaves Generadas (solo las tres primeras y la última, para ejemplificar):
    - Subclave Ronda 0: `000102030405060708090a0b0c0d0e0f`
    - Subclave Ronda 1: `d6aa74fdd2af72fadaa678f1d6ab76fe`
    - Subclave Ronda 2: `b692cf0b643dbdf1be9bc5006830b3fe`
    - Subclave Ronda 10: `39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32`

A.3 Rondas del Cifrado

Cada ronda en AES-128 (excepto la última) incluye las operaciones SubBytes, ShiftRows, MixColumns y AddRoundKey. La última ronda omite MixColumns.

A.3.1 Ronda Inicial (AddRoundKey)

- Estado Inicial: `00112233445566778899aabbccddeeff`
- Subclave Ronda 0: `000102030405060708090a0b0c0d0e0f`
- Resultado Tras AddRoundKey: `00102030405060708090a0b0c0d0e0f`

A.3.2 Rondas 1 a 9

Cada una de estas rondas aplica las siguientes operaciones en secuencia:

1. SubBytes:
    - Se aplica una tabla S-box a cada byte del estado para sustituirlo por otro valor.
    - Estado Tras SubBytes (Ejemplo): `63cab7040953d051cd60e0e7ba70e18c`
2. ShiftRows:
    - Las filas del estado se rotan hacia la izquierda: la segunda fila se rota una posición, la tercera dos, y la cuarta tres.
    - Estado Tras ShiftRows (Ejemplo): `6353e08c0960e104cd70b751bacad0e7`
3. MixColumns:
    - Se transforma cada columna mediante multiplicación en un campo finito, mezclando los datos.
    - Estado Tras MixColumns (Ejemplo): `5f72641557f5bc92f7be3b291db9f91a`
4. AddRoundKey:
    - El estado se combina con la subclave correspondiente a la ronda.
    - Subclave Ronda N: (por ejemplo, para la ronda 1: `d4d1c6f87c839d87caf2b8bc11f915bc`)
    - Estado Tras AddRoundKey: `d4d1c6f87c839d87caf2b8bc11f915bc`

A.3.3 Ronda 10 (Última Ronda)

En la última ronda, solo se aplican SubBytes, ShiftRows y AddRoundKey, omitiendo MixColumns.

1. SubBytes: `d436d70c6b36b8a1c20b628db231a7c3`
2. ShiftRows: `d4bcd91606b87d4b86cb26be8a09d1c2`
3. AddRoundKey (Subclave 10):
    i. Subclave Ronda 10: `3925841d02dc09fbdc118597196a0b32`
    ii. Estado Final: `3925841d02dc09fbdc118597196a0b32`

A.4 Resultado Final

Después de la décima ronda, el texto cifrado resultante es: Texto Cifrado (en hexadecimal): `3925841d02dc09fbdc118597196a0b32`.

Apéndice B

Operaciones de Montgomery en Curve25519

Este apéndice detalla las operaciones adicionales utilizadas en Curve25519, además de la suma y el doble de puntos. Estas operaciones son fundamentales para el funcionamiento eficiente del protocolo Diffie-Hellman y su implementación práctica.

B.1 Suma de Puntos

La suma de puntos en Curve25519 se lleva a cabo utilizando el algoritmo *Montgomery Ladder*, que es un enfoque eficiente para realizar multiplicaciones escalares. Este método permite realizar cálculos de suma de puntos de manera más segura y rápida, al evitar el uso directo de la forma tradicional de suma y duplicación.

La técnica se basa en una secuencia de operaciones que alternan entre duplicar el punto y sumar otros puntos, minimizando así la dependencia del valor del escalar. Esto hace que el algoritmo sea resistente a ataques de tiempo.

Supongamos que queremos calcular P+Q, donde:

- P = (x_1, y_1) = (9,10)
- Q = (x_2, y_2) = (12,14)

Usando la fórmula para la suma de puntos en coordenadas homogéneas:

$$R = (x_3, y_3) = \left(\frac{(x_1y_2 + y_1x_2)}{z_1z_2}, \frac{(y_1y_2 + x_1x_2)}{z_1z_2}\right)$$

Si asumimos que z_1 = z_2 = 1 (coordenadas cartesianas), los cálculos serían:

1. Calcular $x_3 = \frac{(9 \cdot 14 + 10 \cdot 12)}{1 \cdot 1}$ = 126

2. Calcular $y_3 = \frac{(10 \cdot 14 + 9 \cdot 12)}{1 \cdot 1}$ = 168

Entonces, el resultado de la suma de puntos es R = (126, 168).

B.3 Doble de Puntos

El doble de puntos es otra operación esencial, utilizada para calcular 2P. Esta operación se realiza aprovechando las propiedades algebraicas específicas de las curvas elípticas de Montgomery.

Los métodos de doble de puntos están diseñados para ser más rápidos que el método tradicional, permitiendo una ejecución eficiente de operaciones en la curva.

Para calcular 2P: Supongamos que P = (9,10).

1. Calcular la pendiente λ = $\frac{3x_1^2 + a}{2y_1}$ (donde a es una constante de la curva), asumiendo a = 0, nos sale que λ son 12 '15.
2. Calcular $x_3 = \lambda^2 - 2x_1 = 12'15^2$ - 2 · 9 = 129'6225
3. Calcular $y_3 = \lambda\,(x_1 - x_3) - y_1$ = $12'15 \cdot (9 - 129'6225) - 10$ = -1475'568

El resultado del doble de puntos es 2P = $(x_3\,, y_3)$.

B.3 Multiplicación Escalar

La multiplicación escalar es una operación central en la criptografía de curvas elípticas, que calcula el resultado de multiplicar un punto en la curva por un escalar. Se implementa a través del algoritmo de Montgomery, combinando la suma de puntos y el doble de puntos de manera eficiente, siendo esencial para establecer claves en el protocolo Diffie-Hellman.

Para calcular kP donde k = 3 y P = (9, 10):

1. Calcular 2P usando la función anterior de doble de puntos explicada en la sección B.2.
2. Luego, sumamos el punto original P al resultado de 2P usando la operación de suma de puntos explicada en B.1.

El resultado final será 3P = P + 2P.

B.4 Reducción Modular

La reducción modular es necesaria para asegurar que los resultados de las operaciones aritméticas en Curve25519 permanezcan dentro del rango del campo de la curva. Se utilizan algoritmos eficientes para realizar la reducción, que minimizan el tiempo de cálculo y aseguran que los resultados estén dentro de los límites de la curva.

Supongamos que tenemos un resultado z = 1000 y necesitamos reducirlo módulo p (donde p = $2^{255} - 19$): z mod p = z si z < p; de lo contrario, z = z − kp para el mayor k tal que z − kp ≥ 0.

Si z = 1000: Como 1000 < p, entonces z mod p = 1000.

B.5 Conversión de Coordenadas

La conversión de coordenadas es necesaria para cambiar entre diferentes representaciones de un punto en la curva, siendo crucial para la implementación de operaciones eficientes.

Tipos de coordenadas:

- Coordenadas Homogéneas: Se utilizan para facilitar cálculos en el espacio de la curva.
- Coordenadas Cartesianas: Se utilizan en la representación final de los puntos, pero requieren conversiones.

Para convertir un punto de coordenadas homogéneas (x, y, z) a coordenadas cartesianas: (x′,y′) = $\left(\frac{x}{z}, \frac{y}{z}\right)$

Supongamos que tenemos P = (9,10,1): $(x', y') = \left(\frac{9}{1}, \frac{10}{1}\right) = (9, 10)$