Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Estudio de cifrados para comunicaciones 5G y 6G

Study of encryption for 5G and 6G communications

Gianmarco Corbo

La Laguna, *31* de *Mayo* de 2023

Dña. **Jezabel Miriam Molina Gil**, con N.I.F. 78507682-B profesora Ayudante Doctor de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

C E R T I F I C A (N)

Que la presente memoria titulada:

“Estudio de cifrados para comunicaciones 5G y 6G”

ha sido realizada bajo su dirección por D. **Gianmarco Corbo**,

con N.I.E. Y3069132A.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a día de mes de año

Licencia

Resumen

Este trabajo presenta un análisis comparativo de tres algoritmos de cifrado utilizados en comunicaciones de redes: SNOW-V, su mejora SNOW-Vi y Rocca. Estos cifrados desempeñan un papel crucial en la protección de la confidencialidad y seguridad de la información transmitida a través de redes móviles.

En primer lugar, se examina el cifrado SNOW-V, que es un cifrado de flujo para redes 5G, como también SNOW-Vi, una mejora del cifrado SNOW-V. Se examinan sus modificaciones realizadas y se evalúa su impacto en la seguridad y el rendimiento.

Posteriormente, se introduce Rocca, un algoritmo de cifrado recientemente propuesto para su implementación incluso en redes 6G. Se explora la estructura y el funcionamiento de Rocca, así como sus ventajas y desafíos en comparación con SNOW-V y SNOW-Vi.

Palabras clave: Cifrado, 5G, Rocca, SNOW, Redes

Abstract

Comparative analysis of three encryption algorithms used in network communications: SNOW-V, its improvement SNOW-Vi, and Rocca. These encryptions play a crucial role in protecting the confidentiality and security of information transmitted through mobile networks.

First, the SNOW-V encryption is examined, which is a stream cipher for 5G networks, as well as SNOW-Vi, an enhancement of the SNOW-V encryption. Their modifications are examined, and their impact on security and performance is evaluated.

Subsequently, Rocca, a recently proposed encryption algorithm for implementation even in 6G networks, is introduced. The structure and operation of Rocca are explored, as well as its advantages and challenges compared to SNOW-V and SNOW-Vi.

Keywords: Cipher, 5G, Rocca, SNOW, Network

Índice general

Capítulo 1 - Introducción **7**
1.1 Líneas introductorias 7
1.2 Marco teórico 7
1.3 Objeto de estudio 8
Capítulo 2 - Análisis del algoritmo SNOW-V y SNOW-Vi **9**
2.1 LFSR 10
2.2 FSM 11
2.3 Inicialización 12
2.4 Mejora SNOW-Vi 13
Capítulo 3 - Análisis del algoritmo Rocca **15**
3.1 Ronda de actualización de estado 15
3.2 Inicialización 16
3.3 Procesado de AD 17
3.4 Cifrado 17
3.5 Finalización 18
Capítulo 4 - Ejemplo de flujo de datos **19**
4.1 SNOW-V/SNOW-Vi 20
4.2 Rocca 23
Capítulo 5 - Implementación software **26**
5.1 Implementación de SNOW-V 26
5.2 Implementación de SNOW-Vi 30
5.3 Implementación de Rocca 32
Capítulo 6 - Pruebas y comparativa de los resultados **36**
Capítulo 7 - Conclusiones y líneas futuras **40**
Capítulo 8 - Summary and Conclusions **41**
Capítulo 9 - Presupuesto **42**
Capítulo 10 - Apéndice 1: Algoritmo AES **43**
Capítulo 11 - Apéndice 2: Instrucciones SIMD **46**

Capítulo 1 - Introducción

1.1 Líneas introductorias

En el actual panorama de las comunicaciones móviles, la seguridad de la información es una de las principales prioridades debido al crecimiento de los ataques cibernéticos y la exposición de datos potencialmente sensibles de los usuarios. Para esto, la criptografía es una herramienta esencial para proteger la privacidad y la integridad de los datos en las redes móviles.

Entre los diversos algoritmos criptográficos, el cifrado Rocca y el cifrado SNOW-V (con su mejora SNOW-Vi) destacan por ser dos soluciones eficientes y efectivas en la protección de la información en entornos de redes móviles.

1.2 Marco teórico

Con la introducción del 3G se adoptó como estándar el algoritmo de cifrado por bloques conocido como KASUMI, utilizado en el estándar UMTS [13] (Sistema Universal de Telecomunicaciones Móviles). Aunque KASUMI fue ampliamente utilizado, se identificaron vulnerabilidades teóricas y prácticas en su diseño, lo que llevó a la búsqueda de algoritmos de cifrado más robustos para las generaciones futuras.

Con el advenimiento del 4G LTE (Long Term Evolution), se introdujeron los algoritmos AES-128, SNOW-3G y ZUC. AES es ampliamente considerado como uno de los algoritmos de cifrado más seguros y eficientes disponibles actualmente y, de hecho, se ha implementado también en sucesivos cifrados que se han ido desarrollando a lo largo de los siguientes años como parte de sus estructuras.

La llegada de las redes 5G trajo consigo nuevos desafíos de seguridad [7], a necesidad de soportar un mayor número de dispositivos y aplicaciones conectadas, así como la creciente dependencia de la infraestructura de red en la nube debido a una serie de cambios fundamentales en la estructura del sistema, ya que muchos de los nodos entre cada enlace han sido sustituidos por nodos virtuales. Los investigadores y expertos en seguridad han trabajado en el desarrollo de nuevos algoritmos de cifrado adecuados para las redes 5G, menos dependientes de la arquitectura hardware y que tienen también en cuenta el nuevo requisito de seguridad de 256 bits para las claves [9].

A medida que las redes 5G continúan desplegándose a nivel global, ya se está

investigando y explorando el terreno de las redes 6G. La seguridad en las redes 6G se considera un desafío aún mayor debido a la proliferación prevista de dispositivos IoT (Internet de las Cosas) y la creciente necesidad de soportar aplicaciones de misión crítica, como la telemedicina y la conducción autónoma. En esta etapa inicial, se están llevando a cabo también investigaciones para identificar posibles algoritmos de cifrado cuántico, que podrían ofrecer una mayor seguridad frente a los ataques cuánticos futuros [4].

1.3 Objeto de estudio

En este trabajo de fin de grado se realizará un análisis comparativo y exhaustivo entre dos cifrados prominentes: Rocca y SNOW-V. Además se incluirá una evaluación detallada de la mejora de este último, conocida como SNOW-Vi. Este estudio tiene como objetivo brindar una comprensión completa de las fortalezas y debilidades de estos algoritmos criptográficos.

En primer lugar, se llevará a cabo un análisis teórico minucioso que abordará las características, la estructura y el funcionamiento de Rocca y SNOW-V, proporcionando una comprensión profunda de sus propiedades y capacidades.

A continuación, se realizará una comparativa práctica en la cual se implementarán los códigos correspondientes a los tres cifrados mencionados: Rocca, SNOW-V y SNOW-Vi. Este enfoque práctico permitirá realizar mediciones precisas de las velocidades de cada cifrado y evaluar su rendimiento efectivo.

Estos resultados proporcionarán información valiosa sobre la eficiencia y eficacia de los cifrados Rocca, SNOW-V y SNOW-Vi, permitiendo identificar sus fortalezas y posibles áreas de mejora.

Capítulo 2 - Análisis del algoritmo SNOW-V y SNOW-Vi

SNOW-V es un algoritmo de cifrado de flujo, un tipo de algoritmo criptográfico que cifra los datos a medida que se transmiten, operando sobre un flujo continuo de bits en lugar de bloques de datos fijos [12]. En esencia, un cifrado de flujo genera una secuencia de bits pseudoaleatorios que se combina con los datos originales utilizando una operación XOR para producir el texto cifrado.

A grandes rasgos, el funcionamiento básico de un algoritmo de cifrado de flujo es el siguiente:

1. Generación del flujo de cifrado: El generador de flujo de clave utiliza la clave secreta para producir un flujo de bits pseudoaleatorios, que se conoce como flujo de cifrado o secuencia de clave. Esta secuencia de bits debe ser suficientemente larga y aparentemente aleatoria para evitar que se pueda predecir o deducir fácilmente.

2. Operación XOR: El flujo de cifrado generado se combina bit a bit mediante una operación XOR con los datos originales que se desea cifrar. Devuelve un resultado de 1 cuando los bits de entrada son diferentes y un resultado de 0 cuando los bits de entrada son iguales. Esto implica que al aplicar la operación XOR dos veces a un mismo bit, se recupera el valor original del bit.

3. Obtención del texto cifrado: El resultado de la operación XOR es el texto cifrado, que se transmite a través del canal de comunicación. Para descifrar los datos en el receptor, se sigue el mismo proceso utilizando la misma clave secreta para generar el flujo de cifrado y aplicando la operación XOR al texto cifrado recibido.

SNOW-V sigue también este esquema, utilizando para la generación del flujo de cifrado una estructura basada en dos registros de desplazamiento de retroalimentación lineal (LFSR, linear-feedback shift register) y una máquina de estados finitos (FSM, finite-state machine).

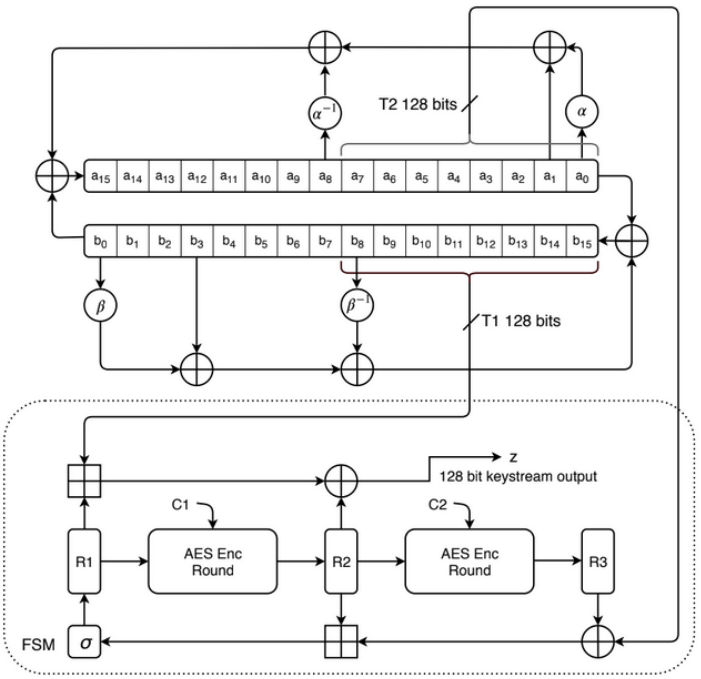

Figura 1. Esquema del algoritmo SNOW-V

2.1 LFSR

En la parte superior de la Figura 1, se pueden observar los dos LFSR (Lineal Feedback Shift Register), denominados LFSR-A y LFSR-B. Estos registros, fundamentales en el funcionamiento del sistema, tienen un tamaño de 256 bits cada uno y están divididos en 16 celdas de 16 bits cada una.

Cada celda tiene la capacidad de almacenar un valor comprendido entre 0 y 65535, representado en forma binaria. Esta amplia gama de valores permite una gran variedad de combinaciones posibles, lo que contribuye a la generación de secuencias pseudoaleatorias robustas.

Aunque ambos LFSR comparten la misma estructura, es importante destacar que tienen diferentes valores de semilla y polinomios de retroalimentación. Los polinomios de retroalimentación desempeñan un papel crucial en la generación de nuevos bits a partir de los existentes en los registros. Estos polinomios se utilizan para seleccionar qué bits se emplearán en la generación de los nuevos bits, combinándolos mediante puertas lógicas XOR.

En el caso de LFSR-A y LFSR-B, las nuevas secuencias generadas son resultado de los siguientes polinomios respectivamente:

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+1)} + \alpha^{-1} a^{(t+8)} \quad mod\ g^{A}(\alpha)$$

$$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+3)} + \beta^{-1} b^{(t+8)} \quad mod\ g^{B}(\beta)$$

Donde $a_i^{(t)}$, 0 ≤ i ≤ 15, son los valores de las celdas en el LFSR-A en el momento *t* y $b_i^{(t)}$, 0 ≤ i ≤ 15, son los valores de las celdas en el LFSR-B en el momento *t.*

$g^A(X)$ y $g^B(X)$ son los dos polinomios que generan los valores de los dos LFSR y están definidos por:

$$g^A(x) = x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1$$

$$g^B(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^8 + x^6 + x^5 + x + 1$$

α y β son respectivamente las raíces de $g^A(X)$ y $g^B(X)$, mientras que α^{-1} y β^{-1} son sus inversas.

Estos polinomios se han diseñado cuidadosamente para garantizar la generación de secuencias pseudoaleatorias con propiedades deseables, como la falta de patrones repetitivos y la imprevisibilidad. Así, los LFSR-A y LFSR-B se convierten en componentes cruciales para la generación de secuencias criptográficas seguras y confiables en el sistema en cuestión.

En cada ciclo de reloj del sistema, se lleva a cabo un proceso clave que implica la extracción de información de los LFSR-A y LFSR-B para la generación de dos bloques de 128 bits cada uno, conocidos como T1 y T2. Específicamente, se toman las 8 celdas menos significativas del LFSR-A y las 8 celdas más significativas del LFSR-B, y se almacenan en sus respectivos bloques.

Una vez completada esta etapa, se inicia la actualización de los LFSR. Es importante destacar que, gracias a las instrucciones SIMD (Single Instruction, Multiple Data, más información en el Apéndice 2) disponibles en los procesadores de generaciones más recientes, es posible lograr una eficiente generación de nuevas secuencias pseudoaleatorias. Esto se logra mediante la aplicación de la retroalimentación de 8 celdas nuevas utilizando una sola instrucción SIMD.

Este enfoque optimizado permite realizar todas las operaciones necesarias para la actualización de los LFSR en un solo ciclo de reloj, logrando así un importante incremento en el rendimiento del algoritmo, ya que se reducen los tiempos de procesamiento necesarios para la generación de las nuevas secuencias.

2.2 FSM

Ahora centrémonos en la estructura de la Máquina de Estados Finitos (FSM), ubicada en la mitad inferior del esquema representado en la Figura 1. Esta FSM toma los dos bloques T1 y T2 provenientes de la etapa de los LFSR como entradas y genera una secuencia de clave de 128 bits como salida.

Para llevar a cabo esta tarea, se utilizan tres registros de 128 bits: R1, R2 y R3. En el diagrama, el símbolo ⊕ representa la operación XOR realizada a nivel de bits.

Además, se emplea la notación de "suma paralela" para indicar que se realizan cuatro sumas módulo 2^{32} de manera simultánea en cada subpalabra de 32 bits. Esto implica que se suman las partes de 32 bits de las palabras de 128 bits, teniendo en cuenta el acarreo, pero sin propagar el acarreo de una palabra de 32 bits más baja a una más alta.

Los registros R2 y R3 se actualizan mediante una única ronda de cifrado AES (Advanced Encryption Standard), cuyo funcionamiento detallado se puede encontrar en el Apéndice 1. Para llevar a cabo esta actualización, se utilizan los valores de dos constantes: C1 y C2, que se establecen en cero. Estas constantes se utilizan como "round keys" o claves de ronda para la ejecución de la ronda de cifrado AES.

Esta estructura bien definida en la FSM contribuye a la generación de una secuencia de clave segura y robusta, brindando una capa adicional de protección en el algoritmo criptográfico en cuestión.

Las fórmulas a través de las cuales se actualizan los registros son las siguientes:

$$
\begin{aligned}
R1^{(t+1)} &= \sigma(R2^{(t)} \boxplus_{32} (R3^{(t)} \oplus T2^{(t)})), \\
R2^{(t+1)} &= AES^{R}(R1^{(t)}, C1), \\
R3^{(t+1)} &= AES^{R}(R2^{(t)}, C2).
\end{aligned}
$$

Donde σ es una permutación de bytes definida por:

σ = [0, 4, 8, 12, 1, 5, 9, 13, 2, 6, 10, 14, 3, 7, 11, 15].

Esto significa que, por ejemplo, el byte 8 se moverá a la posición 2 o que también el byte 1 se moverá a la posición 4.

La secuencia de 128 bits *z,* que será el output del algoritmo, se obtiene según esta expresión:

$$
z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}.
$$

2.3 Inicialización

La fase de inicialización es la que se ejecuta antes de empezar a cifrar o a descifrar el mensaje y su importancia reside en evitar posibles deducciones de la clave que se podrían verificar al usar directamente los valores de la clave en el LFSR.

Cuando iniciamos el algoritmo, se le pasan como parámetros una clave *K* de 256 bits y un vector de inicialización *IV*.

Los 16 bytes menos significativos de la clave *K* pasan a ser los 16 bytes más significativos del LFSR-A, mientras que los 16 bytes menos significativos son los valores del vector de inicialización.

Para el caso del LFSR-B, los 16 bytes más significativos se inicializan como los 16 bytes más significativos de la clave *K*, mientras que los 16 bytes menos significativos se inicializan todos a 0.

Una vez inicializados ambos LFSR, se ejecuta el algoritmo 16 veces, con la única

diferencia de que no se produce ningún output *z*, ya que este último valor retroalimenta los 16 bytes más significativos del LFSR-A con una operación XOR.

Cuando llega a la penúltima iteración, se aplica una operación XOR entre el registro R1 y los 16 bytes menos significativos, mientras que en la última iteración se aplicaría dicha operación al registro R1 con los 16 bytes más significativos de la clave.

2.4 Mejora SNOW-Vi

SNOW-Vi se basa en la misma estructura de SNOW-V, pero introduce mejoras significativas en términos de rendimiento gracias a unos pocos cambios puntuales.

Dichos cambios son:

1. Modificación de las funciones de actualización de los LFSR:

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+7)} \, mod \, g^{A}(\alpha)$$
$$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+8)} \, mod \, g^{B}(\beta)$$

2. Los valores del registro T2 ahora se obtienen de la parte más alta del LFSR-A:

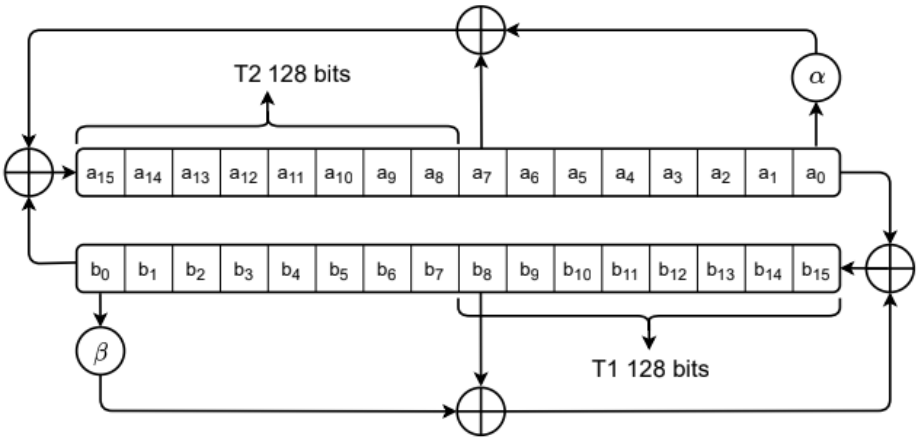

Figura 2. Esquema del nuevo LFSR de SNOW-Vi

3. Nuevos polinomios generadores:

$$g^{A}(x) = x^{16} + x^{14} + x^{11} + x^{9} + x^{6} + x^{5} + x^{3} + x^{2} + 1 \in \mathbb{F}_2[x],$$
$$g^{B}(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^{10} + x^{7} + x^{2} + x + 1 \in \mathbb{F}_2[x].$$

Estos cambios mejoran considerablemente la velocidad de cifrado y también su seguridad, gracias al menor número de operaciones a realizar y a un mejor efecto

de dispersión de bits.

Llamamos las 2 palabras de 128 bits del primer LFSR (A1, A2), mientras que las 2 palabras de 128 bits del segundo serían (B1, B2). Con esta notación, se verifican las siguientes ecuaciones para SNOW-V y SNOW-Vi:

SNOW-V	SNOW-Vi
$T1^{(t)} = B1^{(t)},$ $T1^{(t-1)} = B1^{(t-1)} = B0^{(t)},$ $T2^{(t)} = A0^{(t)},$ $T1^{(t+1)} = B1^{(t+1)}$ $B1^{(t+1)} = A0^{(t)} \oplus f_{\beta}(B0^{(t)}, B1^{(t)})$	$T1^{(t)} = B1^{(t)},$ $T1^{(t-1)} = B1^{(t-1)} = B0^{(t)},$ $T2^{(t)} = A1^{(t)},$ $T1^{(t+1)} = B1^{(t+1)}$ $B1^{(t+1)} = A0^{(t)} \oplus f_{\beta}(B0^{(t)}, B1^{(t)})$

Estas ecuaciones son relevantes en las tres expresiones del cálculo del output del algoritmo durante los ciclos *t, t-1* y *t+1*:

$$z^{(t-1)} = (AES_R^{-1}(\hat{R2}) \boxplus_{32} T1^{(t-1)}) \oplus AES_R^{-1}(\hat{R3}),$$
$$z^{(t)} = (\hat{R1} \boxplus_{32} T1^{(t)}) \oplus \hat{R2},$$
$$z^{(t+1)} = (\sigma(\hat{R2} \boxplus_{32} (\hat{R3} \oplus T2^{(t)})) \boxplus_{32} T1^{(t+1)}) \oplus AES_R(\hat{R1}),$$

Si sustituimos aquí las equivalencias de $T1^{(t)}, T1^{(t-1)}, T1^{(t+1)}$ y $T2^{(t)}$ para el caso de SNOW-V, podemos notar como los valores de $B0^{(t)}$, $B1^{(t)}$ y $A0^{(t)}$ se repiten dos veces y no aparece nunca $A1^{(t)}$, creando así cierto ruido sesgado que podría dar pie a una posible vulnerabilidad, aunque estamos hablando de un ruido muy bajo que prácticamente no puede dar lugar a un ataque lineal.

Con el cambio de SNOW-Vi en cuanto a donde se obtienen los valores de T2, ahora se usarán las cuatro palabras de 128 bits en las ecuaciones, por lo que el anterior ruido disminuye de manera considerable, mejorando aún más la seguridad del algoritmo.

Capítulo 3 - Análisis del algoritmo Rocca

El principal objetivo de Rocca es cumplir con los requisitos de los sistemas 6G tanto en rendimiento como en seguridad, logrando una velocidad de cifrado/descifrado de más de 100 Gbps y proporcionando seguridad de 256 bits contra ataques de recuperación de claves.

Rocca es un cifrado basado en AES con una clave de tamaño 256 bits, pero, a diferencia de SNOW-V/SNOW-Vi, nos encontramos ante un cifrado por bloques y no por flujo.

Estos algoritmos operan sobre bloques fijos de datos en lugar de cifrar cada byte o bit individualmente [6]. Cada bloque de datos se trata como una unidad indivisible y se procesa mediante una serie de operaciones criptográficas.

El tamaño del bloque en los cifrados por bloques varía dependiendo del algoritmo utilizado, pero los tamaños más comunes son 64 bits (8 bytes) y 128 bits (16 bytes). Algunos algoritmos también pueden admitir tamaños de bloque más grandes. El tamaño del bloque tiene implicaciones en la seguridad y el rendimiento del cifrado.

Cuando el tamaño del bloque es grande, hay más bits de información disponibles para el cifrado, lo que hace que sea más difícil para un atacante deducir cualquier patrón o realizar un ataque de fuerza bruta para descifrar el mensaje.

Por otro lado, a medida que el tamaño del bloque aumenta, el tiempo de procesamiento requerido para cifrar o descifrar un mensaje aumenta proporcionalmente. Esto significa que es necesario encontrar un equilibrio entre un tamaño lo suficientemente grande para ser considerado seguro pero que tenga también un rendimiento eficiente.

3.1 Ronda de actualización de estado

El estado de Rocca, que vamos a llamar S, está formado por 8 bloques de 16 bytes cada uno, que van desde S[0] hasta S[7], siendo respectivamente el primer bloque y el último bloque.

La función que actualiza el estado S la definimos como una ronda $R(S, X_0, X_1)$, donde se le pasan como parámetros el estado S y dos bloques que denominamos X_0 y X_1, y se define por las siguientes expresiones:

$$
\begin{aligned}
S^{new}[0] &= S[7] \oplus X_0, \\
S^{new}[1] &= AES(S[0],\ S[7]), \\
S^{new}[2] &= S[1] \oplus S[6], \\
S^{new}[3] &= AES(S[2],\ S[1]), \\
S^{new}[4] &= S[3] \oplus X_1, \\
S^{new}[5] &= AES(S[4],\ S[3]), \\
S^{new}[6] &= AES(S[5],\ S[4]), \\
S^{new}[7] &= S[0] \oplus S[6].
\end{aligned}
$$

Donde AES(X_0, X_1) es una ronda del algoritmo AES.

La ejecución de Rocca se divide en un total de cuatro fases en donde se hace uso de la anterior función para cada una de ellas:

1. Inicialización
2. Procesado de AD
3. Cifrado
4. Finalización

3.2 Inicialización

Al inicializar Rocca hay que pasarle como parámetros la clave de 256 bits $K_0||K_1 \in F_2^{128} \times F_2^{128}$, un nonce *N* de 128 bits, los datos asociados *AD* y el mensaje a cifrar *M*.

Se cargan unos determinados valores en cada bloque del estado, siguiendo las siguientes expresiones:

$$
\begin{aligned}
S[0] &= K_1, \\
S[1] &= N, \\
S[2] &= Z_0, \\
S[3] &= Z_1, \\
S[4] &= N \oplus K_1, \\
S[5] &= 0, \\
S[6] &= K_0, \\
S[7] &= 0.
\end{aligned}
$$

Donde Z_0 y Z_1 son dos bloques constantes Z_0 = 428a2f98d728ae227137449123ef65cd y Z_1 = b5c0fbcfec4d3b2fe9b5dba58189dbbc.

Tras este paso se ejecutan 20 rondas de actualización *R*(S, Z_0, Z_1) y se pasa a la fase de procesado de AD.

3.3 Procesado de AD

Esta fase se salta en caso de que no haya datos asociados.

En caso de que sí los haya, se ejecuta $d - 1$ veces la ronda de actualización *R*(S, AD_i^0, AD_i^1), donde $d = \frac{|AD|}{256}$.

3.4 Cifrado

Esta fase se salta en caso de que no haya mensaje que cifrar.

En caso de que sí hay mensaje, se ejecuta $m - 1$ veces en un bucle las operaciones para el cálculo del texto cifrado *C* y una ronda de actualización *R*(S, M_i^0, M_i^1), donde $m = \frac{|M|}{256}$.

El cálculo del texto cifrado sigue las siguientes expresiones:

$$C_i^0 = AES(S[1], S[5]) \oplus M_i^0,$$

$$C_i^1 = AES(S[0] \oplus S[4], S[2]) \oplus M_i^1.$$

Si el último bloque de *M* está incompleto y su longitud es de *b* bits, es decir $0 < b < 256$, el último bloque de *C* se truncará a los primeros *b* bits.

3.5 Finalización

En esta última fase se ejecutarán 20 rondas de actualización del estado *S R(S,* $|AD|$, $|M|$*)*, para después mostrar la tag formada por los valores resultantes de *S*.

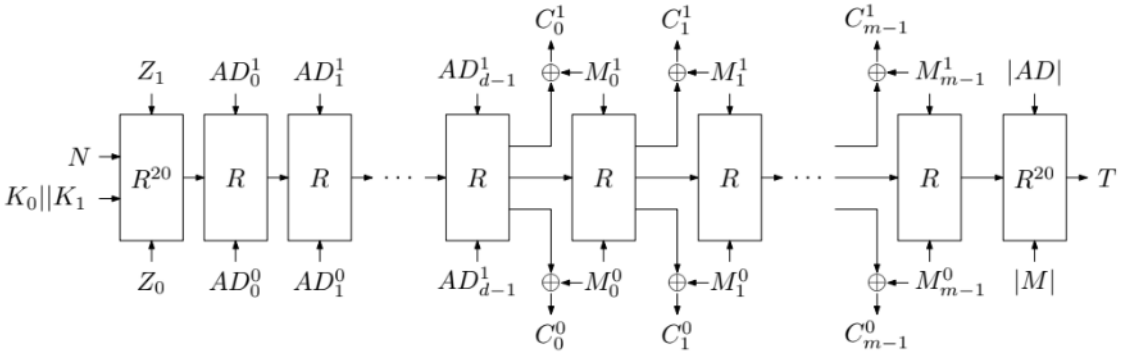

Figura 3. Esquema de la ejecución de Rocca

En la Figura 3 podemos observar cómo se suceden las fases que se han listado, mostrando así el flujo de datos de la ejecución del programa y las entradas y salidas que va generando.

Capítulo 4 - Ejemplo de flujo de datos

En este apartado se presenta un ejemplo descriptivo del flujo del funcionamiento de los cifrados SNOW-V/SNOW-Vi y Rocca con la finalidad de que se entienda mejor su funcionamiento.
Con motivo de no alargar demasiado esta sección solo se mostrará una iteración de SNOW-V, mencionando en qué puntos sería diferente con SNOW-Vi, mientras que para Rocca se mostrará la carga de los valores en los estados en la fase de inicialización y su primera ronda de actualización.

En el caso de SNOW-V/SNOW-Vi se usarán los siguientes valores para los parámetros:

- Clave:
  0x 50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
  0a 1a 2a 3a 4a 5a 6a 7a 8a 9a aa ba ca da ea fa

- IV:
  0x 01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10

Mientras que para el flujo de datos de Rocca serán:

- Clave:
  0x 01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef
  01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef

- AD:
  0x 01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef
  01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef

- Nonce:
  0x 01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef

4.1 SNOW-V/SNOW-Vi

Lo primero que se empieza a ejecutar es la fase de inicialización, que obtiene como parámetros la clave y el vector de IV.
Como se ha dicho anteriormente, los 16 bytes menos significativos de la clave serán los 16 bytes más significativos del LFSR-A, mientras que sus 16 bytes menos significativos son los valores del vector de inicialización.
Para el caso del LFSR-B, los 16 bytes más significativos se inicializan como los 16 bytes más significativos de la clave y los 16 bytes menos significativos se inicializan todos a 0.
Por lo tanto, tras cargar la clave y el vector de IV en los LFSR, ambos tendrían los siguientes valores:

- LFSR-A:

a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
5051	5253	5455	5657	5859	5a5b	5c5d	5e5f	0123	4567	89ab	cdef	fedc	ba98	7654	3210

- LFSR-B:

b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0a1a	2a3a	4a5a	6a7a	8a9a	aaba	cada	eafa	0000	0000	0000	0000	0000	0000	0000	0000

También inicializamos los tres registros de la FSM a 0:

- R1: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- R2: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
- R2: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Ahora se ejecuta el algoritmo de cifrado 16 veces, retroalimentando los 16 bytes más significativos del LFSR-A combinándolos con XOR junto al output z.

Vamos a mostrar cómo sería la primera iteración del algoritmo tras terminar la fase de inicialización, para ilustrar con más claridad las operaciones que se realizan.
Los valores del LFSR-A y el LFSR-B después de la fase de inicialización serán los siguientes:

- LFSR-A:

a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
41cc	bfe4	db70	c424	fd46	36e3	7a95	cc05	9e6c	640e	ae62	525f	7d81	8bd0	92c0	ffd9

- LFSR-B:

b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
e7e1	16e8	c907	6e3b	13b2	4108	6a6a	9787	492b	ee5a	5765	4aac	7dbf	8a8b	071e	56ea

Mientras que los tres registros R1, R2 y R3 serían:

R1 = 5d 7d 71 bd ee 49 b3 8d 9c 3e 4e 37 0b 41 68 32
R2 = ee de 62 5e 3c d7 37 07 91 ac 6d 1d 51 ca 3a b3
R3 = 5f 5a 0a 01 30 c3 05 1f d8 93 3a 74 bd d1 6a a7

Empezamos cargando en el registro T1 los 16 bytes más significativos del LFSR-B:

T1 = e7 e1 16 e8 c9 07 6e 3b 13 b2 41 08 6a 6a 97 87

Se realizan las operaciones para generar el output del cifrado *z*:

$$z^{(t)} = (R1^{(t)} \boxplus_{32} T1^{(t)}) \oplus R2^{(t)}.$$

1. Se hacen cuatro sumas módulo 2^{32} de manera simultánea en cada subpalabra de 32 bits entre R1 y T1, dando como resultado

   tmp = 44 5f 88 a5 b7 51 21 c9 af f0 8f 3f 75 ab ff b9

2. Se realiza una operación XOR entre *tmp* y R2, obteniendo así *z*.

   z = aa 81 ea fb 8b 86 16 ce 3e 5c e2 22 24 61 c5 0a

Ahora comienza la fase de actualización de la FSM:

1. Cargamos los 16 bytes menos significativos del LFSR-A en T2 en el caso de SNOW-V, mientras que para SNOW-Vi se cargarán los 16 bytes más significativos.

   T2 = 9e 6c 64 0e ae 62 52 5f 7d 81 8b d0 92 c0 ff d9

2. Calculamos en un registro temporal *tmp* el resultado de las operaciones que usaremos para la actualización del registro R1. Primero aplicamos XOR entre R3 y T2:

   tmp = $R3 \oplus T2$ = c1 36 6e 0f 9e a1 57 40 a5 12 b1 a4 2f 11 95 7e

   Ahora llevamos a cabo cuatro sumas módulo 2^{32} de manera simultánea en cada subpalabra de 32 bits entre R2 y *tmp*, dando como resultado:

   tmp = af 15 d1 6d da 78 8f 47 36 bf 1e c2 80 db cf 31

3. Actualizamos el registro R3, aplicando una ronda del algoritmo AES sobre el registro R2:

   R3 = $AES_R(R2)$ = 13 1d e9 90 bd 11 89 87 ea 49 ee d7 a0 fa 14 bc

4. Actualizamos el registro R2, aplicando una ronda del algoritmo AES sobre el registro R1:

   R2 = $AES_R(R1)$ = 13 1d e9 90 bd 11 89 87 ea 49 ee d7 a0 fa 14 bc

5. Finalmente, mezclamos los bytes según la expresión de σ para así obtener el registro R1 actualizado:

   R1 = σ(*tmp*) = af da 36 80 15 78 bf db d1 8f 1e cf 6d 47 c2 31

Por último, se empieza la fase de actualización del LFSR, según la siguiente expresión para SNOW-V:

$$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+1)} + \alpha^{-1} a^{(t+8)} \quad mod\ g^{A}(\alpha)$$

$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+3)} + \beta^{-1} b^{(t+8)} \ mod\ g^{B}(\beta)$

Mientras que para SNOW-Vi sería:

$a^{(t+16)} = b^{(t)} + \alpha a^{(t)} + a^{(t+7)} mod\ g^{A}(\alpha)$

$b^{(t+16)} = a^{(t)} + \beta b^{(t)} + b^{(t+8)} \ mod\ g^{B}(\beta)$

Obteniendo así estos valores en ambos registros:

- LFSR-A:

a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
b656	509a	b30b	f181	fa1b	9a02	6e95	000e	41cc	bfe4	db70	c424	fd46	36e3	7a95	cc05

- LFSR-B:

b_{15}	b_{14}	b_{13}	b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0402	ce70	dfc4	15cd	0a61	8a0f	bf21	8364	e7e1	16e8	c907	6e3b	13b2	4108	6a6a	9787

4.2 Rocca

En la fase de inicialización, cargaremos los valores iniciales en cada bloque del estado S:

$S[0] = K_1$ = 01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef

$S[1] = N$ = 01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef

$S[2] = Z_0$ = 42 8a 2f 98 d7 28 ae 22 71 37 44 91 23 ef 65 cd

$S[3] = Z_1$ = b5 c0 fb cf ec 4d 3b 2f e9 b5 db a5 81 89 db bc

$S[4] = N \oplus K_1$ = 00 00 00 00 00 00 00 00 01 23 45 67 89 ab cd ef

$S[5] = 0$ = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

$S[6] = K_0$ = 01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef

$S[7] = 0$ = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Se ejecuta 20 veces la ronda de actualización de estado usando como parámetros

las dos constantes Z_0 y Z_1. Para poder ilustrar su funcionamiento, mostramos las operaciones de la primera iteración:

$S^{new}[0]$= S[7] ⊕ Z0 = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
⊕
42 8a 2f 98 d7 28 ae 22 71 37 44 91 23 ef 65 cd
=
42 8a 2f 98 d7 28 ae 22 71 37 44 91 23 ef 65 cd

$S^{new}[1]$ = AES(S[0], S[7]) = AES(01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef,
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00)
=
d6 6a d8 51 4b d2 11 3a 6a c1 6f 75 9e c5 57 b3

$S^{new}[2]$ = S[1] ⊕ S[6] = 01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef
⊕
01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef
=
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

$S^{new}[3]$ = AES(S[2], S[1]) = AES(42 8a 2f 98 d7 28 ae 22 71 37 44 91 23 ef 65 cd,
01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef)
=
59 6f 67 6f ef 47 9c 79 0b 07 ae 36 88 1d 82 3a

$S^{new}[4]$ = S[3] ⊕ Z1 = b5 c0 fb cf ec 4d 3b 2f e9 b5 db a5 81 89 db bc
⊕
b5 c0 fb cf ec 4d 3b 2f e9 b5 db a5 81 89 db bc
=
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

$S^{new}[5]$ = AES(S[4], S[3]) = AES(00 00 00 00 00 00 00 00 01 23 45 67 89 ab cd ef,
b5 c0 fb cf ec 4d 3b 2f e9 b5 db a5 81 89 db bc)
=
43 22 33 39 16 a4 f0 30 4b 84 8b 4e d2 f7 f7 b3

$S^{new}[6]$ = AES(S[5], S[4]) = AES(00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00,
00 00 00 00 00 00 00 00 01 23 45 67 89 ab cd ef)

=

8e 66 f9 d1 91 8f d9 00 b7 c7 b1 15 81 7a a0 80

$S^{new}[7]$ = S[0] ⊕ S[6] = 01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef

⊕

01 23 45 67 89 ab cd ef 01 23 45 67 89 ab cd ef

=

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Después de ejecutarse 19 iteraciones más, obtenemos el siguiente bloque de estado *S* y terminaríamos la fase de inicialización:

$S[0]$= 1e 53 0b 46 9e 68 a5 9b 91 66 62 3a b6 32 1d 56

$S[1]$ = d2 f6 c1 7a 84 8b 99 55 f3 93 fd 98 ab 09 d2 66

$S[2]$ = 95 22 b4 ba e8 ed ab 3d 62 01 42 d8 7e b4 55 65

$S[3]$ = a1 c1 8d 43 d4 b0 23 83 7e 91 4b 7d 26 e8 c6 10

$S[4]$ = f1 01 99 94 d4 d8 cd c9 2b 11 f1 97 de 51 22 75

$S[5]$ = 24 c9 f6 0d 61 cf 6e 2b 94 8f 39 66 11 41 d3 1b

$S[6]$ = ee 40 de 2e 2b fe 45 1a ab 72 83 38 88 bd 34 51

$S[7]$ = a1 06 a7 80 2e ba 6a 2e 95 bd 03 5c ff 13 7e 85

Capítulo 5 - Implementación software

Para la implementación software de SNOW-V, SNOW-Vi y Rocca se ha optado por emplear el código especificado al final de los papers de cada uno de ellos y, posteriormente, añadir una parte del código para realizar las pruebas de velocidad.

El código que se ha implementado se puede consultar en el siguiente enlace:

https://github.com/Gia-ui/tfg.git

5.1 Implementación de SNOW-V

SNOW-V está diseñado para funcionar de forma muy rápida en un entorno puramente de software, y para ello se han utilizado el conjunto de instrucciones SIMD actualmente disponibles en los procesadores de nuevas generaciones. Esta implementación está codificada con las instrucciones intrínsecas de los procesadores Intel, pero existen también implementaciones similares para otras marcas de CPU.

La parte FSM de SNOW-V es la más sencilla de las dos estructuras a la hora de implementarse, utilizando los registros de 128 bits *__m128i* [11] y la función intrínseca *_mm_aesenc_si128()* [8] para la ejecución del algoritmo AES optimizado para registros de 128 bits.

Para las sumas aritméticas en paralelo se ha utilizado *_mm_add_epi32()* y la permutación de 16 bytes σ se obtiene usando *_mm_shuffle_epi8().*

```
C/C++
struct SnowV256 {
    __m128i R1, R2, R3; // FSM

...

// FSM Update
```

```
        __m128i R3new = _mm_aesenc_si128(R2, _snowv_zero);
        __m128i R2new = _mm_aesenc_si128(R1, _snowv_zero);
        R1 = _mm_shuffle_epi8(_mm_add_epi32(R2, _mm_xor_si128(R3, T2)),
                              _snowv_sigma);
        R3 = R3new;
        R2 = R2new;
        return z;
}
```

Ahora, para la parte del LFSR, se almacena el contenido de los dos registros de 256 bits en unas variables de tipo *__m256i* que se han llamado *hi* y *lo*, de la siguiente manera, y representan las diferentes partes altas y bajas de las dos estructuras LFSR de la siguiente manera:

lo[127 ... 0 bits] = {a7, . . . , a0}
hi[127… 0 bits] = {a15, . . . , a8}
lo[255 ... 128 bits] = {b7, . . . , b0}
hi[255…128 bits] = {b15, . . . , b8}

```
C/C++
struct SnowV256 {
   __m256i hi, lo;     //LFSR
...
}
```

Para poder actualizar los dos LFSR, solo necesitamos calcular nuevos valores para un solo registro, el *hi,* con *hi_new=update(lo, hi),* ya que que los nuevos valores del registro *lo* son una copia del anterior registro *hi*.

El polinomio generador $g^A(\alpha)$ se puede representar con el valor *0x990f*.

La multiplicación de *x* por *α* en f se hace de la siguiente manera: primero desplazamos *x*<<1 y luego, dependiendo del bit en la posición 15 del *x* original, hacemos un xor con el resultado con *gA* (que corresponde a *0x990f*).

Esto se consigue de la siguiente manera:

```
mul_alpha(uint16 x, uint16 gA) := (x<<1) xor ( ((signed int16)x >> 15) and gA)
```

La condición de si hacer la operación con xor o no, se implementa con la ayuda de la máscara de 16 bits *((signed int16)x >> 15)*, donde se crea a partir del desplazamiento aritmético de *x* a la derecha de 15 posiciones, que resulta en la

propagación del bit de signo de la posición 15, formando una máscara 0xffff en caso de que el bit 15 sea 1, o 0x0000 en caso de que sea 0.

Esa filosofía se aplica también al vector combinado de 256 bits *lo = (b7, . . . , b0, a7, . . . , a0)* para multiplicar la primera mitad por α y la parte alta por β al mismo tiempo y para la multiplicación de *hi* por α −1 y β −1, pero usando la instrucción *_mm256_srai_epi16(),* que realiza un desplazamiento aritmético a la derecha de 16 enteros de 16 bits.

```C/C++
// LFSR Update
        __m256i mulx = _mm256_xor_si256(
            _mm256_slli_epi16(lo, 1),
            _mm256_and_si256(_snowv_mul, _mm256_srai_epi16(lo, 15)));

        __m256i invx = _mm256_xor_si256(
            _mm256_srli_epi16(hi, 1),
            _mm256_sign_epi16(_snowv_inv, _mm256_slli_epi16(hi, 15)));

        __m256i hi_old = hi;

        hi = _mm256_xor_si256(
            _mm256_xor_si256(
                _mm256_blend_epi32(_mm256_alignr_epi8(hi, lo, 1 * 2),
                                   _mm256_alignr_epi8(hi, lo, 3 * 2), 0xf0),
                _mm256_permute4x64_epi64(lo, 0x4e)),
            _mm256_xor_si256(invx, mulx));

        lo = hi_old;
```

Para realizar las pruebas de velocidad de cifrado se ha creado una función *main* donde se han declarado e inicializado las variables necesarias. Se crea una estructura llamada *cipher* de tipo *SnowV256*, y se asigna el valor cero a la clave *key* y el vector de inicialización *iv*.

```C/C++
int main() {
    struct SnowV256 cipher;
    unsigned char key[32] = {0}, iv[16] = {0};

    cipher.keyiv_setup(key, iv);

...
}
```

A continuación, se genera un mensaje de prueba, de tamaño 16 kb, y se llena con valores aleatorios utilizando la función *rand()*.

Luego, se inicia un bucle para realizar múltiples pruebas de cifrado y medir el tiempo requerido. El número de pruebas está definido por la constante *numTests*.

Dentro del bucle, se utiliza la función *clock_gettime* para obtener el tiempo de inicio de la prueba en la variable start. Esta función utiliza el reloj *CLOCK_MONOTONIC* para obtener una medida precisa y estable del tiempo.

A continuación, se cifra el mensaje utilizando el algoritmo *SnowV256*. Se itera sobre el mensaje en bloques de 16 bytes, generando un flujo de clave y realizando una operación XOR entre cada bloque de texto plano y el flujo de clave. El resultado se almacena en un vector llamado *ciphertext*. Después de cifrar el mensaje, se obtiene el tiempo de finalización de la prueba utilizando nuevamente la función *clock_gettime* y se almacena en la variable *end*.

Se calcula el tiempo transcurrido restando el tiempo de inicio del tiempo de finalización utilizando la función *timespec_diff,* que calcula la diferencia entre dos estructuras *timespec*.

```C/C++
for (int i = 0; i < numTests; ++i) {
        clock_gettime(CLOCK_MONOTONIC, &start);
        // Cifrar el mensaje
        unsigned char ciphertext[MESSAGE_SIZE];
        for (int j = 0; j < MESSAGE_SIZE; j += 16) {
            __m128i keystream = cipher.keystream();
            __m128i plaintext = _mm_lddqu_si128((__m128i *)(message + j));
            __m128i encrypted = _mm_xor_si128(plaintext, keystream);
            _mm_storeu_si128((__m128i *)(ciphertext + j), encrypted);
        }
        clock_gettime(CLOCK_MONOTONIC, &end);
        long long elapsed = timespec_diff(start, end);
        //Acumula el tiempo total
        totalTime += elapsed;
}
```

Se acumula el tiempo total en la variable *totalTime* sumando el tiempo transcurrido en cada prueba.

Después de finalizar todas las pruebas, se calcula el tiempo promedio dividiendo el tiempo total entre el número de pruebas y se almacena en la variable *averageTime*.

A continuación, se calcula la velocidad de cifrado promedio dividiendo el tamaño del mensaje en bits por el tiempo promedio en segundos. Finalmente, el resultado se almacena en la variable *averageSpeed* y se imprime en la pantalla.

```
C/C++
double averageTime = totalTime / (double)numTests;

// Calcula la velocidad de cifrado promedio en Gbps
double averageSpeed =
       (double)(MESSAGE_SIZE * 8) / (averageTime / 1000000000.0) / 1e9;
```

La parte de código relacionada con las pruebas es prácticamente la misma para la implementación de SNOW-Vi y Rocca, por lo que se omitirá esta última parte de la explicación del código en sus respectivas secciones.

5.2 Implementación de SNOW-Vi

La implementación de SNOW-Vi está sacada de la implementación software optimizada del paper en el que se presentó.

Al ser una versión muy optimizada tiene una estructura bastante diferente a la implementación de SNOW-V.

Es fundamentalmente una mejora de este último, empleando las mismas instrucciones intrínsecas, pero casi todas definidas como macros al comienzo del código y con unas estructuras de datos más óptimas.

La función que se encarga de cifrar/descifrar es *SnowVi_encdec* y es una función inline que toma como parámetros la longitud de los datos a procesar, punteros a los datos de salida *out* y entrada *in*, así como punteros a la clave *key* y el vector de inicialización *iv* utilizados en el algoritmo de cifrado.

Las variables *__m128i A0, A1, B0, B1, R1, R2, R3, T1, T2* son los registros utilizados para almacenar los valores durante cada proceso de cifrado, y que corresponden respectivamente a sus nombres del esquema del cifrado presente en su respectiva sección.

```
C/C++
inline void SnowVi_encdec(int length, unsigned char *out, unsigned char *in,
unsigned char *key, unsigned char *iv) {
```

```
    __m128i A0, A1, B0, B1, R1, R2, R3, T1, T2;

...
}
```

Se inicializan las variables *B0*, *R1* y *R2* a cero utilizando la macro *ZERO()*, cargamos el vector de inicialización *iv* en la variable *A0* como también los primeros 16 bytes de la clave *key* en las variables *R3* y *A1* y los siguientes 16 bytes de la clave en la variable *B1*.

```C/C++
// key /IV loading
    B0 = R1 = R2 = ZERO();
    A0 = LOAD(iv);
    R3 = A1 = LOAD(key);
    B1 = LOAD(key + 16);
```

Posteriormente realizamos la fase de inicialización con los valores de la clave y empezamos la fase de cifrado, donde tenemos un bucle *for* itera desde 0 hasta *length - 16* en incrementos de 16. En cada iteración, se ejecuta la macro *SnowVi_XMM_ROUND(1, i)* para realizar el cifrado en el bloque actual de datos.

```C/C++
#define SnowVi_XMM_ROUND(mode, offset)                                          \
    T1 = B1, T2 = A1;                                                           \
    A1 = XOR(XOR(XOR(TAP7(A1, A0), B0), AND(SRA(A0), SET(0x4a6d))), SLL(A0));   \
    B1 = XOR(XOR(B1, AND(SRA(B0), SET(0xcc87))), XOR(A0, SLL(B0)));             \
    A0 = T2;                                                                    \
    B0 = T1;                                                                    \
    if (mode == 0)                                                              \
        A1 = XOR(A1, XOR(ADD(T1, R1), R2));                                     \
    else                                                                        \
        STORE(out + offset, XOR(ADD(T1, R1), XOR(LOAD(in + offset), R2)));      \
    T2 = ADD(R2, R3);                                                           \
    R3 = AESR(R2, A1);                                                          \
    R2 = AESR(R1, ZERO());                                                      \
    R1 = SIGMA(T2);

...
```

```
// Bulk encryption
    for (int i = 0; i <= length - 16; i += 16) {
        SnowVi_XMM_ROUND(1, i);
    }
```

5.3 Implementación de Rocca

Se define una estructura de contexto llamada *context*, que almacena el estado interno del algoritmo. Dicho estado está representado por un array de 8 valores de 128 bits (__m128i). También se almacenan las longitudes en bytes de los datos de entrada y de los datos asociados.

A continuación, se definen una serie macros y constantes que se utilizan en el código para operaciones de cifrado, actualización de estado y manipulación de datos.

```C/C++
typedef struct Context {
    __m128i state[8]; // state
    size_t sizeM;     // byte length of input data
    size_t sizeAD;    // byte length of associated data
} context;


...
//Macros
...
```

Ahora tenemos 5 funciones que se encargan respectivamente de cada una de las fases de cifrado de Rocca:

- La función *stream_init* inicializa el estado interno del algoritmo a partir de una clave y un valor de nonce proporcionados. Se cargan los valores iniciales del estado, se actualiza el estado local varias veces y se copia el estado actualizado al contexto.

```C/C++
void stream_init(context *ctx, const uint8_t *key, const uint8_t *nonce) {
    __m128i S[S_NUM], M[M_NUM], tmp7, tmp6;
    // Initialize intenal state
    S[0] = _mm_loadu_si128((const __m128i *)(key + 16));
    S[1] = _mm_loadu_si128((const __m128i *)(nonce));
    S[2] = _mm_set_epi32(Z0_3, Z0_2, Z0_1, Z0_0);
    S[3] = _mm_set_epi32(Z1_3, Z1_2, Z1_1, Z1_0);
    S[4] = _mm_xor_si128(S[1], S[0]);
    S[5] = _mm_setzero_si128();
    S[6] = _mm_loadu_si128((const __m128i *)(key));
    S[7] = _mm_setzero_si128();
    M[0] = S[2];
    M[1] = S[3];
    // Update local state
    for (size_t i = 0; i < NUM_LOOP_FOR_INIT; ++i) {
        UPDATE_STATE(M)
    }
    // Update context
    for (size_t i = 0; i < S_NUM; ++i) {
        ctx->state[i] = S[i];
    }
    ctx->sizeM = 0;
    ctx->sizeAD = 0;
}
```

- La función *stream_proc_ad* procesa datos asociados (associated data). Recibe un puntero a los datos asociados y su tamaño en bytes. Copia el estado del contexto a un estado local, procesa los datos asociados en bloques de 32 bytes actualizando el estado local, y finalmente copia el estado actualizado al contexto.

```C/C++
size_t stream_proc_ad(context *ctx, const uint8_t *ad, size_t size) {
    __m128i S[S_NUM], M[M_NUM], tmp7, tmp6;
    // Copy state from context
    for (size_t i = 0; i < S_NUM; ++i) {
        S[i] = ctx->state[i];
    }
    // Update local state with associated data
    size_t proc_size = 0;
    for (size_t size2 = size / BLKSIZE * BLKSIZE; proc_size < size2;
         proc_size += BLKSIZE) {
        LOAD(ad + proc_size, M);
        UPDATE_STATE(M);
    }
```

```
    // Update context
    for (size_t i = 0; i < S_NUM; ++i) {
        ctx->state[i] = S[i];
    }
    ctx->sizeAD += proc_size;
    return proc_size;
}
```

- La función *stream_enc* cifra datos. Recibe un puntero al búfer de salida, un puntero al búfer de entrada y el tamaño de los datos en bytes. Copia el estado del contexto a un estado local, cifra los datos en bloques de 32 bytes actualizando el estado local, guarda los datos cifrados en el búfer de salida y finalmente copia el estado actualizado al contexto.

```C/C++
size_t stream_enc(context *ctx, uint8_t *dst, const uint8_t *src, size_t size) {
    __m128i S[S_NUM], M[M_NUM], C[M_NUM], tmp7, tmp6;
    // Copy state from context
    for (size_t i = 0; i < S_NUM; ++i) {
        S[i] = ctx->state[i];
    }
    // Generate and output ciphertext
    // Update internal state with plaintext
    size_t proc_size = 0;
    for (size_t size2 = size / BLKSIZE * BLKSIZE; proc_size < size2;
         proc_size += BLKSIZE) {
        LOAD(src + proc_size, M);
        XOR_STRM(M, C);
        STORE(C, dst + proc_size);
        UPDATE_STATE(M);
    }
    // Update context
    for (size_t i = 0; i < S_NUM; ++i) {
        ctx->state[i] = S[i];
    }
    ctx->sizeM += proc_size;
    return proc_size;
}
```

- La función *stream_dec* descifra los datos que se le pasan como parametros, y es exactamente igual a la anterior función con la excepción de que en algunas operaciones *M* y *C* se invierten de lugar.

```
C/C++
LOAD(src + proc_size, M);
XOR_STRM(M, C);
STORE(C, dst + proc_size);
```

- La función *stream_finalize* finaliza el proceso de cifrado y genera una etiqueta de autenticación. Copia el estado del contexto a un estado local, actualiza el estado local con la longitud de los datos asociados y los datos de entrada procesados, y finalmente genera la etiqueta realizando una operación XOR en todos los valores del estado local. La etiqueta resultante se guarda en un búfer proporcionado.

```
C/C++
void stream_finalize(context *ctx, uint8_t *tag) {
    __m128i S[S_NUM], M[M_NUM], tmp7, tmp6;
    // Copy state from context
    for (size_t i = 0; i < S_NUM; ++i) {
        S[i] = ctx->state[i];
    }
    M[0] = CAST_U64_TO_M128((uint64_t)ctx->sizeAD << 3);
    M[1] = CAST_U64_TO_M128((uint64_t)ctx->sizeM << 3);
    // Update internal state
    for (size_t i = 0; i < NUM_LOOP_FOR_INIT; ++i) {
        UPDATE_STATE(M)
    }
    // Generate tag by XORing all S [ i ] s
    for (size_t i = 1; i < S_NUM; ++i) {
        S[0] = _mm_xor_si128(S[0], S[i]);
    }
    // Output tag
    _mm_store_si128((__m128i *)tag, S[0]);
}
```

Capítulo 6 - Pruebas y comparativa de los resultados

Para realizar las pruebas se ha ejecutado el código compilado de cada cifrado 50 veces y se ha realizado una media de entre todas las velocidades obtenidas en Gbps (Gigabit por segundo).

Esto se ha repetido en dos entornos diferentes, a los que llamaremos Entorno 1 y Entorno 2.

- Entorno 1: ordenador portátil con un procesador Intel i7-8750H @ 2.20GHz con frecuencia turbo máxima de hasta @ 4.10GHz, 32 GB de RAM y sistema operativo Ubuntu 22.04.

- Entorno 2: ordenador portátil con un procesador Intel I7-7700HQ @ 2.80GHz con frecuencia turbo máxima 3.90 GHz, 16 GB de RAM y sistema operativo Ubuntu 21.04.

Cada algoritmo se ha probado en un solo hilo y con diferentes longitudes del texto plano de entrada.

Para compilar cada uno de los archivos, se ha hecho uso del siguiente comando y parámetros [14]:

gcc -mavx2 -maes -march=native -O3 -o output cifrado.cpp

Como podemos observar, se está compilando con el compilador gcc con una serie de parámetros orientados a la optimización del código:

- *-mavx2:* habilita el conjunto de instrucciones Advanced Vector Extensions 2 (AVX2) al compilar el código fuente. AVX2 es una extensión de la arquitectura x86 que introduce un conjunto de instrucciones vectoriales de 256 bits para mejorar el rendimiento en cálculos numéricos y de punto flotante.

- -maes: habilita el conjunto de instrucciones AES (Advanced Encryption Standard) al compilar el código fuente. El uso de esta opción indica al compilador que debe generar instrucciones de máquina específicas para aprovechar las instrucciones de cifrado AES disponibles en el procesador objetivo. Esto garantiza que el código resultante esté optimizado para utilizar las características de hardware específicas

de AES.

Al aprovechar el conjunto de instrucciones AES, es posible lograr un rendimiento mejorado en operaciones de cifrado y descifrado, lo que es especialmente beneficioso en aplicaciones que requieren un procesamiento rápido y eficiente de grandes volúmenes de datos, como es nuestro caso.

- *-march=native*: es utilizada para indicar al compilador que genere código optimizado específicamente para la arquitectura del procesador en el que se está compilando. Esta opción permite aprovechar al máximo las características y extensiones de instrucciones disponibles en el hardware subyacente.

  El compilador examina el procesador del sistema en el momento de la compilación y determina automáticamente la arquitectura objetivo más adecuada. Así se puede obtener un código binario altamente optimizado que aprovecha las características avanzadas del procesador, como extensiones de instrucciones específicas, conjuntos de registros ampliados y técnicas de optimización específicas de la arquitectura objetivo. Esto puede conducir a mejoras significativas en el rendimiento y la eficiencia del programa.

- *-O3*: es una de las opciones de optimización disponibles en el compilador GCC (GNU Compiler Collection). Indica al compilador que realice optimizaciones agresivas en el código fuente para mejorar el rendimiento del programa resultante.

  La opción -O3 habilita el nivel de optimización 3, el cual es el nivel más alto de optimización disponible en GCC. Esta opción aplica una serie de transformaciones y técnicas de optimización complejas para generar un código más eficiente y rápido, aunque a costa de un mayor tiempo de compilación. Es también indicada para aplicaciones donde el rendimiento es crítico y el tiempo de compilación no es una preocupación principal.

En el caso del Entorno 1, las especificaciones hardware son bastante parecidas a las utilizadas en las pruebas hechas en los papers de SNOW-V y SNOW-Vi, mientras que las del paper de Rocca se realizaron en una CPU de dos generaciones más adelantada. Debido a esto, probablemente, y como se puede observar en la Tabla 1, la consecuente cercanía de los resultados obtenidos en las pruebas realizadas con los dos primeros cifrados y los resultados obtenidos en sus propuestas originales. En el caso de Rocca se obtuvieron unos valores ligeramente más bajos respecto a los que figuran en los resultados obtenidos en su respectivo paper.

Algoritmo de cifrado	Tamaño del mensaje						
	16384	8192	4096	2048	1024	256	64
SNOW-V (C++)	55.35	52.66	51.27	46.93	41.39	29.42	19.37
SNOW-Vi (C++)	70.41	67.23	63.79	54.21	45.29	21.80	20.07
Rocca (C++)	156.45	119.10	106.61	93.26	79.41	47.87	16.04

Tabla 1. Resultados de las pruebas realizada sobre los tres cifrados en el Entorno 1

Para el Entorno 2, disponemos de un hardware un poco más viejo, ya que se quería probar cada cifrado con unos componentes precedentes a los que se usaron en las pruebas de los autores. Como era de esperar, los datos resultantes han sido algo inferiores a los del Entorno 1, pero aún así se sigue manteniendo un buen rendimiento.

Los resultado obtenidos figuran en la tabla 2:

Algoritmo de cifrado	Tamaño del mensaje						
	16384	8192	4096	2048	1024	256	64
SNOW-V (C++)	48.70	47.96	47.21	46.33	45.87	33.16	27.83
SNOW-Vi (C++)	66.51	64.57	61.43	53.33	47.14	24.86	27.31
Rocca (C++)	147.13	141.86	123.96	117.04	106.79	68.16	22.71

Tabla 2. Resultados de las pruebas realizada sobre los tres cifrados en el Entorno 2

Al analizar los resultados en ambos entornos, podemos apreciar que son consistentes con los hallazgos presentados en las respectivas propuestas originales [1] [2] [3], lo que demuestra una correcta medición de las velocidades de cada algoritmo.

Como era de esperar, SNOW-V muestra una velocidad más lenta en comparación con los otros dos algoritmos. En el Entorno 1, alcanza una velocidad promedio de hasta 55.35 Gbps, mientras que en el Entorno 2, logra un rendimiento de aproximadamente 48.70 Gbps. Aunque SNOW-V puede no ser el más veloz, sigue siendo una opción viable para aplicaciones que no requieren una alta velocidad de procesamiento.

En cuanto a SNOW-Vi, se puede observar claramente la mejora en velocidad que ha introducido en comparación con su versión anterior. Esta mejora se traduce en un

incremento de casi 20 Gbps, lo que refuerza su posición como una opción más eficiente y rápida dentro de los algoritmos analizados.

Por otro lado, Rocca destaca como el algoritmo más rápido, con una diferencia sustancial con respecto a los demás. Alcanza una velocidad media máxima impresionante de 156.45 Gbps, superando los requisitos tanto para las redes 5G como para las futuras redes 6G. Esta velocidad excepcional coloca a Rocca como una opción sólida y confiable para aplicaciones que demandan un alto rendimiento y una respuesta rápida.

Capítulo 7 - Conclusiones y líneas futuras

La comparativa ha revelado que los cifrados SNOW-V, SNOW-Vi y Rocca cumplen perfectamente con los requisitos de velocidad y seguridad necesarios en entornos 5G y potencialmente 6G. Estos resultados respaldan también la viabilidad de implementarlos en aplicaciones y sistemas que requieran altos niveles de eficiencia en el ámbito de las comunicaciones.

No obstante, es esencial tener en cuenta el apartado de seguridad. SNOW-Vi ha demostrado mejoras en comparación con su predecesor SNOW-V, Rocca también debe someterse a rigurosas evaluaciones y pruebas para garantizar su resistencia a ataques y su solidez frente a posibles debilidades.

En este sentido, se subraya la necesidad de ampliar la investigación en el ámbito de la seguridad criptográfica de Rocca, con el objetivo de profundizar sobre su efectiva seguridad y realizar comparativas más exhaustivas en diversos escenarios de uso. De esta manera, se podrá obtener una comprensión más completa de las capacidades y limitaciones de estos algoritmos, lo que permitirá tomar decisiones más informadas y respaldadas por evidencias en cuanto a su implementación en sistemas y aplicaciones sensibles, ya que resulta esencial destacar que la elección de un cifrado adecuado debe basarse en un enfoque que considere no solo el rendimiento y la velocidad, sino también la seguridad y la resistencia a ataques. Un análisis exhaustivo de estos aspectos, junto con las necesidades y requisitos específicos de cada escenario de uso, sería fundamental para tomar decisiones informadas y asegurar la protección de la información y los sistemas.

Capítulo 8 - Summary and Conclusions

The comparative analysis has revealed that the SNOW-V, SNOW-Vi, and Rocca ciphers meet the required speed and security standards in 5G environments and potentially 6G. These results support the feasibility of implementing them in applications and systems that require high levels of efficiency in the field of communications.

However, it is essential to take into account the security aspect. Although SNOW-Vi has demonstrated improvements compared to its predecessor SNOW-V, Rocca must also undergo rigorous evaluations and testing to ensure its resistance to attacks and its robustness against possible weaknesses.

In this regard, the need to expand research in the field of cryptographic security of Rocca is emphasized, with the aim of gaining a deeper understanding of its effective security and conducting more comprehensive comparisons in various usage scenarios. This will allow for a more complete understanding of the capabilities and limitations of these algorithms, enabling informed decisions supported by evidence regarding their implementation in sensitive systems and applications. It is essential to highlight that the choice of an appropriate encryption should be based on an approach that considers not only performance and speed but also security and resistance to attacks. An exhaustive analysis of these aspects, together with the specific needs and requirements of each usage scenario, would be crucial for making informed decisions and ensuring the protection of information and systems.

Capítulo 9 - Presupuesto

Tipos	Precio	Descripción	Cálculo coste total
Ingeniero Informático	13,72€ / hora	Sueldo promedio por hora de un Ingeniero Informático en España. Es el que se encarga de analizar e implementar el código.	13,72€ x 640 horas de trabajo total = 8780,80€
Ordenador portátil	878€	Coste de la compra de un Ordenador portátil HP OMEN 15-DC1000NS, con un hardware muy similar al que se ha usado para los tests en las propuestas de SNOW-V y SNOW-Vi (Entorno 1)	878€
Coste total	9798,80€	-	8780,80€ + 878€ = 9658,80€

Tabla 3: Resumen de tipos

Capítulo 10 - Apéndice 1: Algoritmo AES

El cifrado AES (Advanced Encryption Standard) [5] es un tipo de cifrado simétrico, ya que utiliza la misma clave tanto para cifrar como para descifrar datos, y se basa en el uso de la red de permutación de sustitución (SPN), donde se aplican múltiples rondas para cifrar los datos. Estas rondas de encriptación son la razón principal por la que AES es tan seguro, ya que se requiere romper un gran número de rondas para comprometer la seguridad del cifrado.

En los cifrados tratados en este trabajo, se hace uso de este algoritmo usando la instrucción *__mm_aesenc_si128*, que es parte del conjunto de instrucciones AES-NI (Advanced Encryption Standard - New Instructions) que se encuentran en las arquitecturas de procesadores modernos. Esta instrucción permite realizar una ronda de cifrado del algoritmo AES en el modo de operación Electronic Codebook (ECB) utilizando una clave de cifrado de 128 bits.

El formato general de la instrucción *__mm_aesenc_si128* es el siguiente:

```C/C++
__m128i mm_aesenc_si128 (__m128i a, __m128i RoundKey)
```

Esta instrucción toma dos operandos de 128 bits (*__m128i*), *a* y *RoundKey*, y devuelve un resultado de 128 bits.

a representa el bloque de texto plano de entrada, mientras que *RoundKey* es la clave de ronda generada por la expansión de clave del algoritmo AES.

El nombre *__mm_aesenc_si128* indica que se trata de una instrucción SIMD que opera en registros de 128 bits utilizando la tecnología AES. Esta instrucción realiza una ronda del algoritmo de cifrado AES sobre los datos almacenados en un registro de 128 bits.

El algoritmo AES se basa en operaciones de sustitución y permutación de bytes, así como en operaciones de mezcla. Una ronda del algoritmo AES consta de cuatro etapas: SubBytes, ShiftRows, MixColumns y AddRoundKey.

La instrucción *__mm_aesenc_si128* realiza estas operaciones en paralelo en los datos almacenados en el registro de 128 bits.

Toma dos operandos: el registro de entrada *a* y la clave de ronda *RoundKey*. Ambos deben ser registros SIMD de 128 bits. La instrucción realiza las siguientes

operaciones:

1. Sustitución de bytes (SubBytes): Aplica una sustitución no lineal a cada byte del registro a utilizando una tabla de búsqueda específica del algoritmo AES. Esta operación realiza una transformación no lineal en los datos para aumentar la seguridad del cifrado.

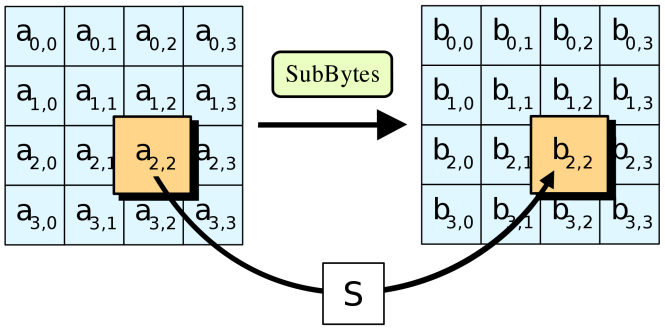

Figura 4. Esquema del funcionamiento de SubBytes

2. Desplazamiento de filas (ShiftRows): Realiza un desplazamiento cíclico de las filas del registro *a*. Cada byte se desplaza a una posición específica dentro de su propia fila. Esta operación asegura que los datos se mezclen y difuminen en la siguiente etapa del algoritmo.

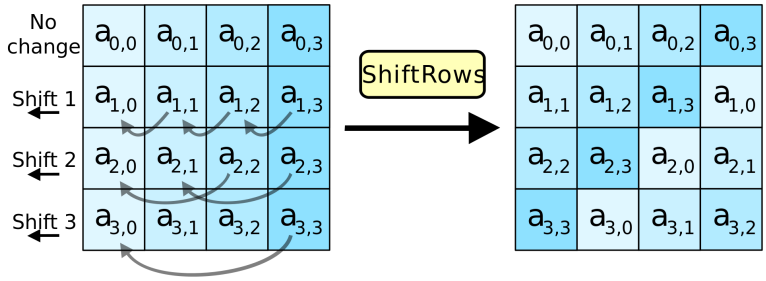

Figura 5. Esquema del funcionamiento de ShiftRows

3. Mezcla de columnas (MixColumns): Realiza una operación de mezcla lineal en cada columna del registro *a*. Cada columna se trata como un polinomio de 4 bytes y se multiplica por una matriz fija. Esta operación mezcla los bytes dentro de cada columna para aumentar la difusión de los datos.

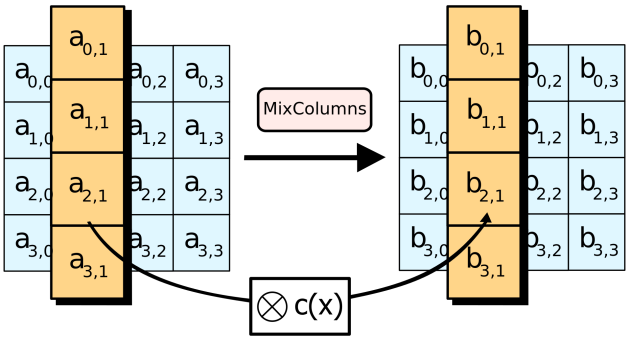

Figura 6. Esquema del funcionamiento de MixColumns

4. Suma de clave de ronda (AddRoundKey): Realiza una operación XOR entre el registro *a* y la clave de ronda *RoundKey*. La clave de ronda se genera a partir de la clave original del algoritmo AES y se utiliza en cada ronda para introducir entropía en el cifrado.

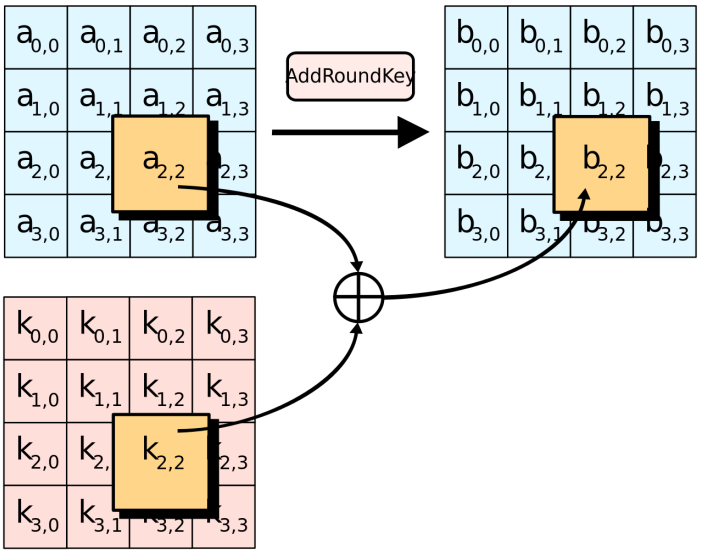

Figura 7. Esquema del funcionamiento de AddRoundKey

Capítulo 11 - Apéndice 2: Instrucciones SIMD

Las instrucciones SIMD (Single Instruction, Multiple Data) son un conjunto de instrucciones que permiten realizar operaciones en paralelo en arquitecturas de procesadores. Estas instrucciones se utilizan para acelerar el procesamiento de datos en aplicaciones que requieren realizar operaciones repetitivas y simultáneas en conjuntos de datos.

La idea principal detrás de las instrucciones SIMD es ejecutar una sola instrucción en múltiples elementos de datos al mismo tiempo. En lugar de procesar un solo dato a la vez, permiten procesar varios datos en paralelo en un solo ciclo de reloj, lo que mejora significativamente el rendimiento y la eficiencia del procesador.

En la mayoría de las arquitecturas de procesadores modernos, las instrucciones SIMD se implementan a través de extensiones de conjunto de instrucciones especiales, como SSE (Streaming SIMD Extensions) o AVX (Advanced Vector Extensions) en los procesadores Intel, o NEON en los procesadores ARM. Estas extensiones proporcionan registros de tamaño ampliado que pueden contener múltiples elementos de datos y operaciones especializadas para realizar cálculos simultáneos en esos elementos.

Cuando se utilizan instrucciones SIMD, los datos se organizan en vectores, que son secuencias contiguas de elementos de datos. Cada elemento del vector se procesa simultáneamente utilizando la misma operación aritmética o lógica. Por ejemplo, si tienes un vector de 8 elementos de datos y deseas sumarlos todos, una instrucción SIMD puede realizar la suma de los 8 elementos en paralelo.

Son especialmente útiles en aplicaciones que involucran procesamiento de señales, gráficos, multimedia, física y otras tareas que se benefician de la capacidad de procesar grandes conjuntos de datos de forma simultánea. Algunos ejemplos comunes de operaciones SIMD incluyen la suma, resta, multiplicación, división, comparación y mezcla de elementos de datos.

Para utilizar instrucciones SIMD, los programas deben estar optimizados y adaptados para aprovechar su capacidad de procesamiento paralelo. Esto se logra utilizando bibliotecas y marcos de desarrollo que proporcionan interfaces de programación de aplicaciones (API) para aprovechar las instrucciones SIMD, o mediante la escritura directa de código en lenguajes que admiten este tipo de instrucciones, como C, C++ o ensamblador.

A continuación, se explicará el funcionamiento de las instrucciones SIMD específicas que se han utilizado en la implementación de los cifrados de este trabajo [10]:

- *_mm_xor_si128(__m128i a, __m128i b):* se utiliza para realizar una operación de XOR (bit a bit) entre dos registros de 128 bits. Esta instrucción toma dos registros como entrada y devuelve un nuevo registro con el resultado de la operación XOR aplicada a cada par de elementos de los registros de entrada *a* y *b*. La función devuelve un nuevo registro *__m128i* que contiene el resultado de dicha operación.

  Se ha usado también una versión para 256 bits llamada *_mm256_xor_si256.*

- *_mm_and_si128(__m128i a, __m128i b)*: se utiliza para realizar una operación lógica AND (bit a bit) entre dos registros de 128 bits. Esta instrucción toma dos registros *a* y *b* como entrada y devuelve un nuevo registro con el resultado de la operación AND aplicada a cada par de elementos de los registros de entrada.

- *_mm_add_epi32(__m128i a, __m128i b)*: se utiliza para realizar una suma de enteros de 32 bits entre dos registros de 128 bits. Es decir, suma cada elemento de los registros a y b, Donde a y b son los registros de 128 bits que se van a sumar, y almacena el resultado en un nuevo registro __m128i.

- *_mm_set1_epi16(short value)*: se utiliza para cargar un valor de 16 bits en todos los elementos de un registro de 128 bits. *value* es el valor de 16 bits que se desea cargar en todos los elementos del registro.

- *_mm_slli_epi16(__m128i a, int imm8)*: se utiliza para realizar un desplazamiento lógico a la izquierda (shift left) de cada elemento de un registro de 128 bits que contiene enteros de 16 bits. Esta instrucción toma dos operandos como entrada: el registro *a* que se va a desplazar y una constante entera de 8 bits *imm8* que especifica la cantidad de bits para el desplazamiento. Devuelve un nuevo registro *__m128i* con los elementos desplazados.

  Se ha usado también una versión para 256 bits llamada *_mm256_slli_epi16.*

- *_mm_srai_epi16(__m128i a, int imm)*: toma un registro de entrada *a* y realiza un desplazamiento aritmético a la derecha de cada elemento de 16 bits en a según el valor inmediato especificado por *imm*. El desplazamiento aritmético a la derecha (SRA) preserva el bit de signo del número, lo que significa que el bit más significativo (MSB) se mantiene igual después del desplazamiento. Cada elemento de 16 bits en el registro se desplaza a la derecha por un número específico de bits, y el valor inmediato proporcionado determina la cantidad de bits que se desplazan.

- *_mm_alignr_epi8(__m128i a, __m128i b, int count)*: toma los registros de entrada a y b, y desplaza los elementos del registro *a* hacia la derecha en *count* bytes, alineándolos con los elementos del registro *b*. Luego, concatena los elementos alineados y devuelve un nuevo registro SIMD con los resultados de la concatenación.

- *_mm_shuffle_epi8(__m128i a, __m128i mask)*: toma el registro de entrada *a* y

aplica la máscara de reordenamiento *mask* para seleccionar y reordenar los elementos del registro *a*. Cada elemento en la máscara especifica la ubicación del elemento correspondiente en el registro de entrada *a* que se copiará al registro resultante. Si el elemento en la máscara es cero, el elemento correspondiente en el registro de entrada se descarta.

El resultado de la función es un nuevo registro SIMD que contiene los elementos reordenados seleccionados según la máscara de reordenamiento.

Es importante tener en cuenta que solo realiza operaciones dentro de los límites de los registros de 128 bits, por lo que la máscara de reordenamiento debe tener la misma longitud que el registro de entrada.

- *_mm_set_epi64x(long long int e1, long long int e0)*: toma dos valores enteros de 64 bits y los coloca en un nuevo registro SIMD de 128 bits. El valor *e1* se coloca en los bits de orden superior del registro SIMD, mientras que el valor *e0* se coloca en los bits de orden inferior.

- *_mm_setzero_si128(void)*: crea y devuelve un nuevo registro SIMD de 128 bits, donde todos los elementos están configurados en cero. Este registro se utiliza comúnmente como un punto de partida para realizar operaciones SIMD, ya que proporciona un valor inicial predefinido y consistente.

- *_mm_loadu_si128(const __m128i* mem_addr)*: carga los datos de la memoria apuntada por *mem_addr* en un nuevo registro SIMD de 128 bits. La instrucción asume que los datos están en una ubicación de memoria arbitraria y no necesariamente alineada a la alineación requerida por la arquitectura SIMD.

- *_mm_storeu_si128(__m128i* mem_addr, __m128i a)*: almacena los datos del registro SIMD a en la ubicación de memoria apuntada por *mem_addr*. Al igual que la función *_mm_loadu_si128*, esta instrucción asume que la ubicación de memoria es arbitraria y no necesariamente alineada a la alineación requerida por la arquitectura SIMD.

- *_mm_lddqu_si128(const __m128i* mem_addr)*: toma como argumento un puntero a la memoria *mem_addr* que contiene los datos y carga los 128 bits de datos sin alineación en un registro SIMD de 128 bits.

  Se ha usado también una versión para 256 bits llamada *_mm256_lddqu_si256*.

- *_mm256_castsi256_si128(__m256i a)*: toma como argumento un registro de 256 bits (*__m256i*) y devuelve un registro de 128 bits. En concreto, esta instrucción extrae los primeros 128 bits del registro de 256 bits y los coloca en un nuevo registro de 128 bits.

  Es útil cuando se necesita utilizar solo la mitad inferior del registro de 256 bits en operaciones que requieren registros de ese tamaño.

- *_mm256_zextsi128_si256(__m128i a)*: toma como argumento un registro de 128 bits y realiza una extensión de ceros para convertirlo en un registro de 256 bits. En otras palabras, los 128 bits del registro de 128 bits se copian en los 128 bits menos significativos del registro de 256 bits, y los 128 bits más significativos del registro de 256 bits se establecen en cero.

- *_mm256_sign_epi16(__m256i a, __m256i b)*: realiza una multiplicación de los elementos de *a* por los elementos de *b* y conserva el signo del resultado. Si el elemento correspondiente de *b* es negativo, el resultado tendrá su signo cambiado; de lo contrario, el resultado mantendrá su signo original.

- *_mm_aesenc_si128(__m128i a, __m128i RoundKey)*: realiza una ronda de cifrado AES en los datos contenidos en el registro *a*, utilizando la clave de ronda especificada en el registro *RoundKey*. La instrucción aplica las operaciones de sustitución, permutación y mezcla necesarias para transformar los datos según el algoritmo AES. Esto permite realizar operaciones de cifrado AES de manera eficiente utilizando instrucciones SIMD.

Bibliografía

- [1]: Ekdahl, Patrik, Thomas Johansson, Alexander Maximov, and Jing Yang. 2019. “A New SNOW Stream Cipher Called SNOW-V”. IACR Transactions on Symmetric Cryptology 2019 (3):1-42. https://doi.org/10.13154/tosc.v2019.i3.1-42.
- [2]: Ekdahl, Patrik, Alexander Maximov, Thomas Johansson, and Jing Yang. ‘SNOW-Vi: An Extreme Performance Variant of SNOW-V for Lower Grade CPUs’. In Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks, 261–72. WiSec ’21. New York, NY, USA: Association for Computing Machinery, 2021. https://doi.org/10.1145/3448300.3467829.
- [3]: Sakamoto, Kosei, Fukang Liu, Yuto Nakano, Shinsaku Kiyomoto, and Takanori Isobe. 2021. “Rocca: An Efficient AES-Based Encryption Scheme for Beyond 5G”. IACR Transactions on Symmetric Cryptology 2021 (2):1-30. https://doi.org/10.46586/tosc.v2021.i2.1-30.
- [4]: Porambage, Pawani, Gürkan Gür, Diana Pamela Moya Osorio, Madhusanka Liyanage, Andrei Gurtov, and Mika Ylianttila. ‘The Roadmap to 6G Security and Privacy’. IEEE Open Journal of the Communications Society PP (05 2021). https://doi.org/10.1109/OJCOMS.2021.3078081.
- [5]: “Advanced Encryption Standard (AES).” 2023. GeeksforGeeks. https://www.geeksforgeeks.org/advanced-encryption-standard-aes/.
- [6]: “Cifrado de bloque: qué es y cómo funciona.” n.d. Ciberseguridad. Accessed June 4, 2023. https://ciberseguridad.com/guias/prevencion-proteccion/criptografia/cifrado-bloque/.
- [7]: “5G PPP Architecture White Paper.” 2022. 5G-PPP. https://5g-ppp.eu/wp-content/uploads/2022/12/6G-Arch-Whitepaper_v1.0-final.pdf
- [8]: Gueron, Shay. n.d. “R) Advanced Encryption Standard (AES) New Instructions Set White Paper.” Intel. Accessed June 4, 2023. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf.
- [9]: Hussain, Alifya. n.d. “5G SDN and NFV. 5G is the 5th generation mobile… | by

Alifya Hussain.” Medium. Accessed June 4, 2023. https://medium.com/@alifyahussain/5g-sdn-and-nfv-e411dbe927b1.

- [10]: “Intel® Intrinsics Guide.” n.d. Intel. Accessed June 4, 2023. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.
- [11]: “__m128i.” 2021. Microsoft Learn. https://learn.microsoft.com/en-us/cpp/cpp/m128i?view=msvc-170.
- [12]: Patricio, Héctor. 2021. “Tipos de algoritmos criptográficos: cifrados de flujo.” The Dojo MX Blog. https://blog.thedojo.mx/2021/12/12/tipos-de-algoritmos-criptograficos-cifrados-de-flujo.html.
- [13]: “TS 135 202 - V7.0.0 - Universal Mobile Telecommunications System (UMTS); Specification of the 3GPP confidentiality and integrity.” n.d. ETSI. Accessed June 4, 2023. https://www.etsi.org/deliver/etsi_ts/135200_135299/135202/07.00.00_60/ts_135202v070000p.pdf.
- [14]: “x86 Options (Using the GNU Compiler Collection (GCC)).” n.d. GCC, the GNU Compiler Collection. Accessed June 4, 2023. https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html.