Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

# Análisis e implementación del SNOW-V

*SNOW-V analysis and implementation*

Óscar Cigala Álvarez

La Laguna, 14 de julio de 2022

Dña. **Molina Gil, Jezabel Miriam**, con N.I.F. 78.507.682-B profesora Ayudante a Doctor adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora.

Dña. **González González, Yanira**, con N.I.F. 78.555.933-P Doctora en Informática, contratada como Técnico de Grado Medio en el Servicio de Sistemas de Información del Complejo Hospitalario Universitario de Canarias, como cotutora.

**CERTIFICAN**

Que la presente memoria titulada:

”Análisis e implementación del SNOW-V”

ha sido realizada bajo su dirección por D. **Óscar Cigala Álvarez**, con N.I.F. 79.073.357-R.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 30 de junio de 2022

Agradecimientos

A mis tutoras Jezabel Molina y Yanira González por ayudarme y hacer posible este Trabajo de Fin de Grado.

A mi familia y a mi pareja por apoyarme siempre durante estos años de carrera.

# Licencia

© Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional.

## Resumen

*El objetivo de este Trabajo de Fin de Grado consiste en realizar un análisis completo del generador de SNOW-V, cuyos autores pretenden que sea implementado como cifrado primitivo en sistemas de telefonía móvil 5G. Para ello, se implementó este cifrado de flujo y se estudiaron diferentes técnicas software para mejorar su eficiencia y cotejarlas con la implementación que viene implícita en el algoritmo y poder sacar conclusiones. Estas posibles mejoras se compararon en dos plataformas con diferentes características, siendo la primera un portátil de gama media y la segunda un IDE online que tiene unas prestaciones altas, para así analizar cómo se comporta el algoritmo en diferentes entornos. Los resultados obtenidos demuestran que la implementación Tradicional de SNOW-V no es la más eficiente, siendo la modificación de Ventanas Deslizantes la óptima en ambas plataformas.*

**Palabras clave:** SNOW-V, cifrado de flujo, implementación software, 5G

## Abstract

*The objective of this Final Degree Project is to carry out a complete analysis of the SNOW-V generator, whose authors intend it to be implemented as the primitive encryption in 5G mobile systems. For this, this stream cipher was implemented and different software techniques were studied to improve its efficiency and compare them with the implementation that is implicit in the algorithm to draw conclusions. These possible improvements were compared on two platforms with different characteristics, the first being a mid-range laptop and the second an online IDE with high performance, in order to analyze how the algorithm behaves in different environments. The results obtained show that the Traditional implementation of SNOW-V is not the most efficient, being the Sliding Windows modification the optimal one in both platforms.*

**Keywords:** SNOW-V, stream cipher, software implementation, 5G

# Índice general


**Capítulo 1 Introducción** . . . . . . . . . . . . . . . . . . . . . . . . . **1**

1.1 La telefonía móvil . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 El cifrado SNOW-V . . . . . . . . . . . . . . . . . . . . . . . . 2

1.3 Objetivos y procedimiento . . . . . . . . . . . . . . . . . . . . . 2

**Capítulo 2 Estado del arte** . . . . . . . . . . . . . . . . . . . . . . . **4**

**Capítulo 3 Descripción del generador de SNOW-V** . . . . . **5**

**Capítulo 4 Implementación software y análisis SNOW-V** . **9**

4.1 Dispositivos y herramientas utilizadas para la evaluación . . . . . . . . . . 9

4.2 Herramientas empleadas para la medición . . . . . . . . . . . . . . . . . 10

**Capítulo 5 Análisis de diferentes técnicas para implementar el LFSR** . . . . . . . . . . . . . . . . . . . . . . . . . **14**

5.1 Implementación Tradicional . . . . . . . . . . . . . . . . . . . . . 15

5.2 Implementación Hardcode . . . . . . . . . . . . . . . . . . . . . . 17

5.3 Implementación de Búfer Circular . . . . . . . . . . . . . . . . . . 19

5.4 Implementación de Ventanas Deslizantes . . . . . . . . . . . . . . . 22

5.5 Implementación de Desdoblamiento de Bucles . . . . . . . . . . . . 24

**Capítulo 6 Análisis de resultados** . . . . . . . . . . . . . . . . . . **25**

6.1 Análisis de resultados con el dispositivo Asus . . . . . . . . . . . . . 26

6.2 Análisis de resultados con Codeanywhere . . . . . . . . . . . . . . . 29

**Capítulo 7 Conclusiones y líneas futuras** . . . . . . . . . . . . . **32**

**Capítulo 8 Summary and conclusions** . . . . . . . . . . . . . . . **34**

**Capítulo 9 Presupuesto** . . . . . . . . . . . . . . . . . . . . . . . . **36**

# Índice de figuras

Figura 1: Esquema general de SNOW-V . . . . . . . . . . . . . . . . . . . . . . . . . 5
Figura 2: Funciones que componen el encriptado AES . . . . . . . . . . . . . . . . . . 7
Figura 3: Llamador y destinatario del Generador de perfiles de rendimiento . . . . . 12
Figura 4: Funciones del Generador de perfiles de rendimiento . . . . . . . . . . . . 13
Figura 5: LFSR de la implementación Tradicional . . . . . . . . . . . . . . . . . . . . 15
Figura 6: LFSR de la implementación Hardcode . . . . . . . . . . . . . . . . . . . . . 17
Figura 7: Funcionamiento de la implementación Búfer Circular . . . . . . . . . . . . 19
Figura 8: Funcionamiento de la implementación de Ventanas Deslizantes . . . . . . 22

# Índice de tablas

Tabla 1: Especificaciones del dispositivo utilizado para la evaluación . . . . . . . . . 9

Tabla 2: Especificaciones disponibles de Codeanywhere . . . . . . . . . . . . . . . . 10

Tabla 3: Rendimiento de las funciones de SNOW-V en el ordenador portátil . . . . . 11

Tabla 4: Rendimiento de las funciones de SNOW-V en Codeanywhere . . . . . . . . 11

Tabla 5: Tiempo de CPU consumido por las funciones . . . . . . . . . . . . . . . . . . 12

Tabla 6: Tiempos de ejecución de lfsr_update empleando el dispositivo Asus . . . . 27

Tabla 7: Tiempos totales de cada implementación empleando el dispositivo Asus . . . 28

Tabla 8: Mejora de eficiencia del total de Ventanas Deslizantes respecto al resto de implementaciones empleando el dispositivo Asus . . . . . . . . . . . . . . . . . . . . 28

Tabla 9: Mejora de eficiencia del LFSR de Ventanas Deslizantes respecto al resto de implementaciones empleando el dispositivo Asus . . . . . . . . . . . . . . . . . . . . 28

Tabla 10: Tiempos totales de cada implementación empleando Codeanywhere . . . 29

Tabla 11: Tiempos de ejecución de lfsr_update empleando Codeanywhere . . . . . 30

Tabla 12: Mejora de eficiencia del total de Ventanas Deslizantes respecto al resto de implementaciones empleando Codeanywhere . . . . . . . . . . . . . . . . . . . . . . 31

Tabla 13: Mejora de eficiencia del LFSR de Ventanas Deslizantes respecto al resto de implementaciones empleando Codeanywhere . . . . . . . . . . . . . . . . . . . . . . 31

Tabla 14: Presupuesto total del proyecto . . . . . . . . . . . . . . . . . . . . . . . . 36

# Capítulo 1

# Introducción

## 1.1 La telefonía móvil

Desde finales del siglo XX hasta la actualidad, se han desarrollado y puesto en marcha una serie de oleadas o generaciones de comunicaciones inalámbricas. Todo comenzó con la segunda generación (2G) a principio de la década de los 90 con la aparición del estándar GSM desarrollado en Europa, que dio los primeros pasos gracias a una buena calidad de voz, dispositivos móviles realmente portátiles, itinerancia de datos y la instauración de un mercado competitivo con multitud de operadores y fabricantes. Unos años más tarde, el 2G evolucionó al 2.5G al implementar un dominio de conmutación de paquetes además del de circuitos. Respecto a los sistemas de cifrado empleados, comenzó con el cifrado de flujo A5/1 y su consiguiente versión A5/2 para garantizar la seguridad de las comunicaciones en telefonía móvil 2G [1].

A medida que el uso de teléfonos 2G se generalizó y las personas comenzaron a utilizar los móviles diariamente, se hizo evidente que la demanda de datos (como el acceso para navegar por internet) estaba creciendo. Además, la experiencia de los servicios de banda ancha fija mostró que también habría una demanda cada vez mayor de mejoras respecto a velocidad de datos. La tecnología 2G era incapaz de satisfacer las crecientes demandas, por lo que la industria comenzó a trabajar en la siguiente generación de tecnología, conocida como tercera generación (3G). Esta nueva versión nace con la premisa de mejorar la capacidad de transmisión de datos para poder ofrecer servicios como la conexión a internet, la descarga de archivos e incluso la videoconferencia. Además, debido a que al final de la etapa del 2G se detectaron ciertas debilidades de seguridad en los cifrados de flujo A5/1 y A5/2, fueron sustituidos por el sistema de cifrado en bloque conocido como Kasumi, naciendo así la tecnología 3G o UMTS que sucedió a la tecnología GSM [2].

Sin embargo, en 2010 el cifrado de bloque fue fácilmente atacado y quedó claro que, en algún momento, las redes 3G se verían abrumadas por el crecimiento de las aplicaciones de uso intensivo de ancho de banda, como la transmisión de medios. En consecuencia, la industria comenzó a buscar tecnologías de cuarta generación (4G) optimizadas para datos, con la promesa de mejoras de velocidad hasta 10 veces superiores a las tecnologías 3G existentes. El 4G se comercializó bajo el estándar *Long Term Evolution* (LTE) usando el algoritmo de confidencialidad UEA2 y el de integridad UIA2 y su versión mejorada *LTE-Advanced* (LTE-A), que empleó EEA1 y EIA1 [3].

A principios de 2018 comenzó a comercializarse la quinta generación (5G), para la cual fue diseñado SNOW-V. Esta nueva tecnología ofrece una tasa de latencia extremadamente baja, pasando de los 40 - 80 milisegundos (ms) que había en el 4G a 1 ms con 5G. Además, el límite de velocidad apunta a tasas de rendimiento que van desde los 20 Gigabits por segundo (Gbps) hasta los 50 Gbps [1]. Más allá de las mejoras, se espera que la tecnología dé rienda suelta a un ecosistema 5G de internet de las cosas (IoT) masivo, donde las redes pueden satisfacer las necesidades de comunicación de miles de millones de dispositivos conectados, con las compensaciones adecuadas entre velocidad, latencia y costo. Respecto a la tecnología, la organización de estandarización 3GPP busca aumentar el nivel de seguridad de EIAx y EEAx de los 128 bits de tamaño de clave a los 256 bits, debido a que este estándar se sigue considerando seguro [4].

## 1.2 El cifrado SNOW-V

SNOW-V es un cifrado de flujo con el objetivo de ser implementado como cifrado principal en sistemas 5G. Se basa en gran medida en SNOW 3G, diseñado como función principal de cifrado en sistemas 3G y 4G. Para alcanzar los nuevos objetivos de 5G y aumentar la flexibilidad de implementación, *Ericsson Research*, junto con la Universidad de Lund han desarrollado este nuevo cifrado llamado SNOW-V, donde la letra V hace referencia a virtualización [5].

Uno de los principales motivos para desarrollar SNOW-V fue satisfacer la demanda de la industria de tener un cifrado muy rápido en un entorno virtualizado, es decir, que sea ejecutado en la nube mediante software, en vez de en los dispositivos mediante hardware. Con este cifrado se pretendió anticiparse al lanzamiento del 5G para evitar futuros problemas de compatibilidad y hacer que las implementaciones de 5G fuesen lo más flexibles posibles [3]. En el caso de que no se hubiera conseguido un cifrado lo suficientemente rápido para el 5G, la seguridad se hubiera convertido en un cuello de botella en las comunicaciones, tirando por la borda todas las promesas que hacía esta nueva tecnología, como velocidad de descarga increíblemente rápida, latencia casi despreciable o mejoras en la industria como, por ejemplo, los coches autónomos, poder operar a pacientes de forma remota, etc.

## 1.3 Objetivos y procedimiento

A lo largo de este Trabajo de Fin de Grado se realizará un estudio completo del cifrado SNOW-V, comenzando por un análisis del mismo y avanzando de manera progresiva por este nuevo cifrado de flujo hasta llegar a su implementación.

En esta implementación, nos centraremos en la búsqueda del método que más tiempo y recursos consume, para así mejorar la eficiencia y velocidad del algoritmo

criptográfico. Esperamos que la función que más tiempo y recursos consuma sea la encargada del LFSR (*Linear Feedback Shift Register*) [6], debido a que este método se ocupa de todo el desplazamiento de ambos LFSRs, incluyendo la operación de retroalimentación que reciben como entrada. A continuación, se implementarán distintas modificaciones con las que poder mejorar el tiempo consumido por este método y un análisis objetivo donde extraer cuál es el más óptimo y el que supone una mayor mejora de la eficiencia [2].

Para llevar esto a cabo, se emplearon dos librerías que calculan el tiempo que tarda una sección de código en ejecutarse, llamadas *ctime* [7] y *chrono* [8]. Ambas fueron probadas en el IDE *Atom* [9] haciendo uso de la PowerShell de Windows y en el IDE online *Codeanywhere* [10]. Para respaldar aún más los resultados, se analizó el tiempo de los procesos de CPU del cifrado mediante el programa *Visual Studio 2022* [11], el cual posee una herramienta llamada *Generador de perfiles de rendimiento* [12] que permite analizar de forma gráfica todos estos aspectos para optimizar al máximo el código.

# Capítulo 2

# Estado del arte

La familia de los cifrados de flujo SNOW nació en el año 2000 cuando SNOW 1.0 fue propuesto [13]. Este primer diseño era muy simple, contando con un LFSR y un FSM. Con el paso del tiempo, se descubrieron varias debilidades, como los ataques por diferenciación (*distinguishing attack*) que requería solamente conocer una salida del generador de longitud $2^{95}$ [2].

Como resultado de estas debilidades, SNOW no fue incluido en el conjunto de algoritmos del proyecto NESSIE. Para solventar estos problemas, los autores propusieron un nuevo cifrado de flujo en 2003 denominado SNOW 2.0 [14], donde aseguraban haber arreglado las inconsistencias y mejorado el rendimiento. Con estos avances se logró que SNOW 2.0 fuese uno de los cifrados de flujo elegidos dentro del estándar ISO/IEC 18033-4 [15]. Además, SNOW 2.0 posee unos principios de diseños similares al cifrado de flujo SOSEMANUK, el cual es uno de los 4 cifrados software elegidos para el portafolia eSTREAM [16].

Tras una evaluación realizada por el *European Telecommunications Standards Institute* (ETSI), el SNOW 2.0 fue modificado para dar paso a SNOW 3G [17], utilizado como uno de los algoritmos estándar para la protección de la integridad y la confidencialidad de los datos a través de los sistemas de comunicación móvil 3G y posteriormente 4G. Uno de los principales objetivos de este nuevo cifrado de flujo fue tener una gran resistencia a ataques de tipo algebraicos [18]. SNOW 3G se elige como el cifrado de flujo para los algoritmos de confidencialidad UEA2 y UIA2, publicado en 2006 por el 3GPP [17], porque permite mayores tasas de datos en dispositivos móviles gracias a su eficiencia cuando es implementado en dispositivos con recursos limitados. Todo esto permitió a SNOW 3G erigirse como un generador resistente a ataques por diferenciación lineal, pero débil a otros que estén basados en sincronización de la caché en tiempo de datos empírico, el cual permite recuperar el estado de inicio completo en segundos sin necesidad de conocer previamente ningún bit [19].

Con todo esto, llegamos al cifrado SNOW-V propuesto en 2019 por los mismos autores junto a la Universidad de Lund [3]. Este algoritmo es una revisión detallada de su versión anterior, enfocada a reforzar la seguridad sin perder la velocidad que ha caracterizado a la familia de los algoritmos SNOW, siendo muy adecuado para su implementación software utilizando instrucciones vectorizadas (SIMD).

# Capítulo 3

# Descripción del generador de SNOW-V

Al igual que ocurría con el diseño de SNOW 3G, el patrón de SNOW-V consta de una parte con dos registros de desplazamiento con retroalimentación lineal o LFSRs y de otra que consiste en una máquina de estados o FSM. En la Figura 1, se puede apreciar el esquema general del cifrado [5].

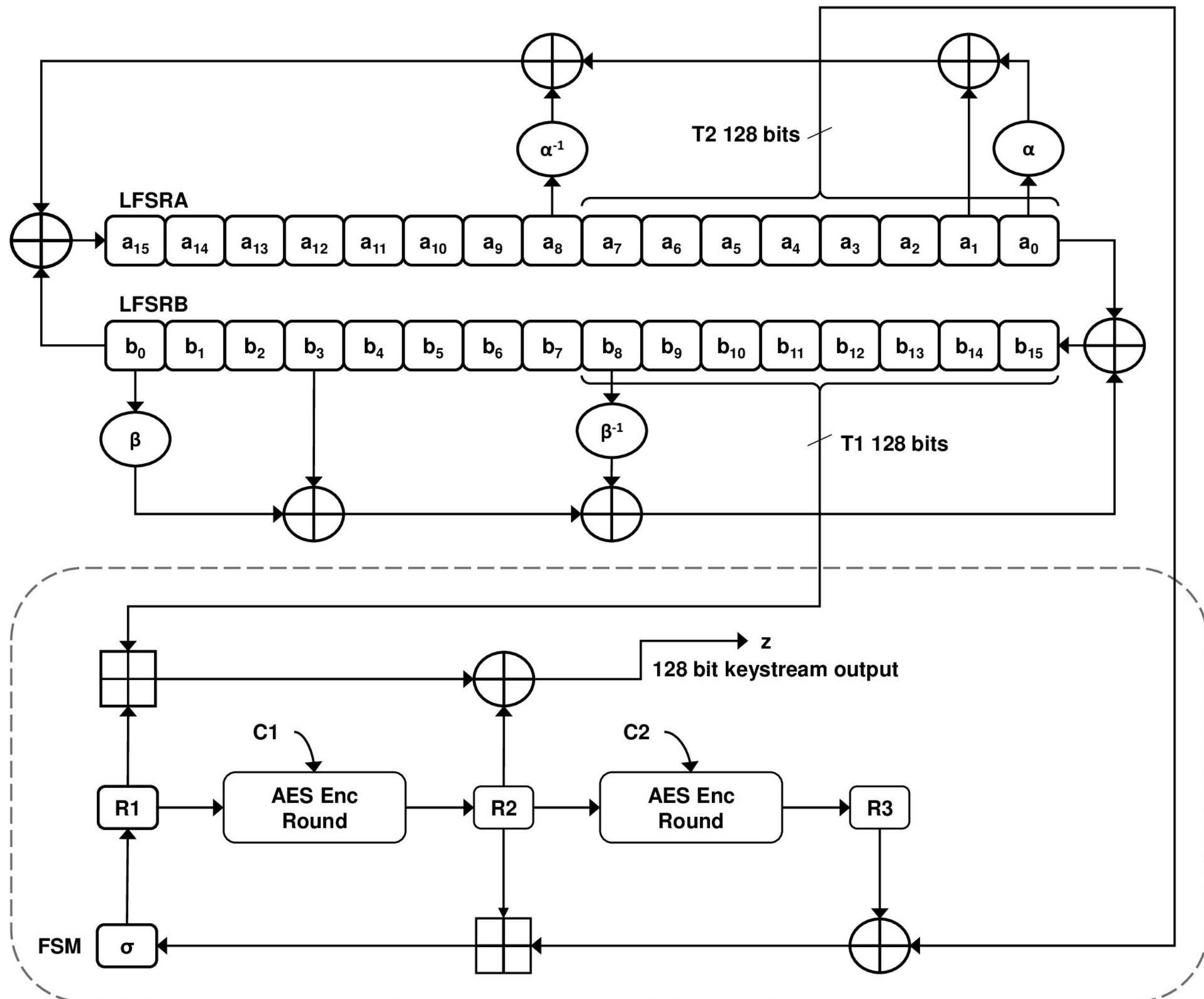


Figura 1: Esquema general de SNOW-V.

Un LFSR es un registro de desplazamiento en el cual la entrada es un bit que proviene de aplicar una función de transformación lineal a un estado anterior [6]. En el caso de SNOW-V, contiene dos registros de desplazamiento en vez de uno, como ocurría en SNOW 3G. Denominaremos a estos **LFSRA** y **LFSRB**, donde la salida que produce uno, sirve para alimentar al otro mediante una operación XOR con otros registros provenientes de otras operaciones. Ambos tienen un tamaño de 16 estados, con 16 bits cada uno, lo que ofrece una seguridad de 256 bits. La nomenclatura que utilizaremos será $\mathbf{a_0, \dots, a_{15}}$ para los estados del LFSRA y $\mathbf{b_0, \dots, b_{15}}$ para los estados del LFSRB.

En cada ciclo de reloj, SNOW-V produce una salida de una palabra de 128 bits (*keystream*), lo que es una mejora sustancial con respecto a los 32 bits que producía su predecesor, y garantiza un buen rendimiento para las arquitecturas actuales de los procesadores. Además, cada LFSR se actualiza 8 veces con cada clock de reloj, lo que permite que con cada pulso estén actualizados los dos valores (**T1** y **T2**) de derivación de 128 bits. Otra ventaja que presenta este cifrado es que los LFSRs fueron revisados para que funcionaran rápidamente con instrucciones **SIMD**, con la finalidad de que fuera enormemente paralelizable y a la vez que mantenía un buen nivel de seguridad [3].

Respecto al polinomio primitivo del LFSRA, este genera los siguientes elementos:

$$g^A(x) = x^{16} + x^{15} + x^{12} + x^{11} + x^8 + x^3 + x^2 + x + 1 \in F_2[x]$$

y el polinomio primitivo del LFSRB genera estos otros:

$$g^B(x) = x^{16} + x^{15} + x^{14} + x^{11} + x^8 + x^6 + x^5 + x + 1 \in F_2[x]$$

La función de retroalimentación del LFSRA viene dada por la ecuación:

$$a_{16} = a_0\alpha \oplus a_1 \oplus a_8\alpha^{-1} \oplus b_0$$

donde $\boldsymbol{\alpha}$ es la raíz de $g^A(x)$ y $\boldsymbol{\alpha^{-1}}$ es la inversa sobre $\mathbf{F}^A_{2^{16}}$. También podemos denotar el LFSRB por:

$$b_{16} = b_0\beta \oplus b_3 \oplus b_8\beta^{-1} \oplus a_0$$

donde $\boldsymbol{\beta}$ es la raíz de $g^B(x)$ y $\boldsymbol{\beta^{-1}}$ es la inversa sobre $\mathbf{F}^B_{2^{16}}$.

Respecto al FSM, recibe como entradas los bloques T1 y T2 que ambos LFSRs producen como salida de palabra. Posee 3 registros de 128 bits cada uno, denominados R1, R2 y R3. En SNOW 3G estos registros eran de 32 bits, por lo que mejora ampliamente la seguridad en esta parte. Además, los registros R2 y R3 son actualizados mediante una única ronda completa de AES, este uso es similar al que se usaba en SNOW 3G con la diferencia del aumento significativo de bits en cada registro. Una ronda completa de AES consta de 4 pasos [20]:

1. Primero se realiza la operación de **SubBytes**, en la cual se produce una sustitución donde cada byte es reemplazado con su recíproco en una tabla de búsqueda.
2. En segundo lugar, viene el paso **ShiftRows**, donde los bytes de cada fila del estado se desplazan cíclicamente hacia la izquierda, siendo este desplazamiento distinto para cada fila. La primera fila no se desplaza, la segunda se desplaza uno a la izquierda, la tercera dos y la cuarta tres.
3. A continuación, se realiza la fase de **MixColumns**, donde se produce una operación de mezclado que opera con las columnas del estado, multiplicando cada una de ellas por un polinomio constante **c(x)**.
4. Por último, se encuentra el paso **AddRoundKey**, donde cada byte del estado se combina mediante una XOR con un byte de C1 o C2, dependiendo de si es el registro R1 o R2.

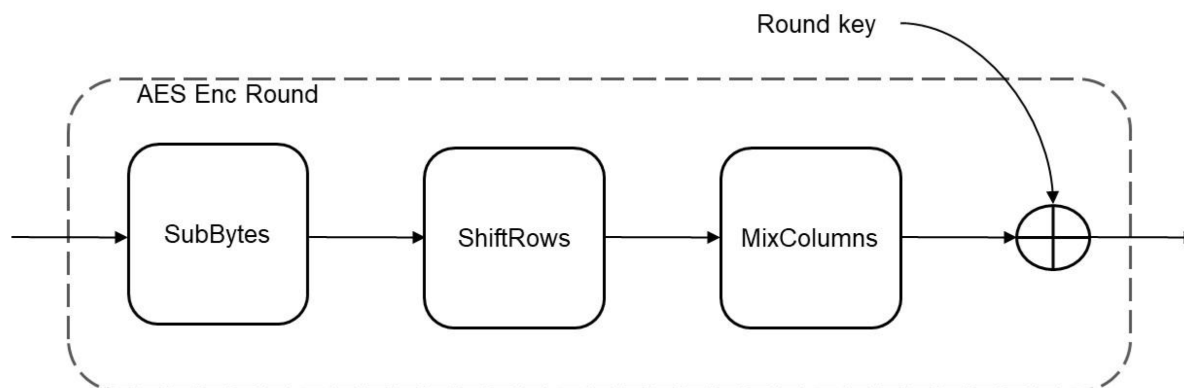


Figura 2: Funciones que componen el encriptado AES.

Otro aspecto importante es que se ha implementado una nueva característica de seguridad llamada modo FP, el cual impide rastrear la clave secreta incluso si el estado interno está comprometido [5]. Por último, se ha añadido otra particularidad en SNOW-V denominada Permutación Sigma, que refuerza aún más la seguridad al modificar implícitamente los primeros 128 bits de la Caja S para que sean diferentes de la segunda. Sigma es representada con **σ** y hace referencia a una permutación de bytes de la forma:

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

Esto debe ser interpretado como que el byte 0 se mueve a la posición 0, el byte 4 se mueve a la posición 1, el byte 8 a la posición 2, etc., constituyendo la posición 0 el byte menos significativo.

Las expresiones de los 3 registros son dadas por las expresiones:

$$R1 = \sigma(R2 \boxplus_{32} (R3 \oplus T2))$$
$$R2 = AES^{R}(R1, C1)$$
$$R3 = AES^{R}(R2, C2)$$

Los valores de C1 y C2 son constantes inicializadas a cero.

Para finalizar este apartado, la salida del cifrado es dada por la expresión:

$$z = (R1 + \boxplus_{32} T1) \oplus R2$$

# Capítulo 4

# Implementación software y análisis de SNOW-V.

Esta sección se centrará en las aplicaciones que han sido empleadas para conseguir ejecutar correctamente el cifrado de flujo SNOW-V, además de en las herramientas utilizadas para analizar las diferentes implementaciones que serán explicadas en el Capítulo 5.

## 4.1 Dispositivos y herramientas utilizadas para la evaluación

Con el objetivo de crear un punto de comparación para futuras investigaciones, la implementación y análisis del código del cifrado SNOW-V fueron ejecutadas en un portátil con Windows 10 Home y empleando el lenguaje C++, ya que el código principal de los propios autores se puede compilar tanto en C como en C++. A la hora de ejecutar las pruebas, estas fueron realizadas siempre en las mismas condiciones, reiniciando el portátil y solo abriendo el IDE donde se ejecutaría el código para conseguir que los datos de todas las implementaciones fueran lo más correctos posibles. Además, se desactivó el modo *Authenticated Encryption with Associated Data* (AEAD), el cual proporciona al cifrado protección de confidencialidad e integridad. Esta opción emplea el fichero ghash que fue eliminado, ya que su uso quedaba fuera del ámbito de este Trabajo de Fin de Grado. Para finalizar, las pruebas fueron realizadas con el *Test Vectors #1* que aparece en el documento de SNOW-V [3].

En particular, el dispositivo utilizado ha sido un **Asus GL502VM-FY213T**, cuyas especificaciones se encuentran en la Tabla 1.

| **Asus GL502VM-FY213T** | | | | |
|---|---|---|---|---|
| Arquitectura | Frecuencia CPU | Cache L1/l2/l3 | RAM | Gráfica |
| 64 bits | i7-7700HQ 2.80GHz | 256 Kb / 1.0 MB / 6.0 MB | 12 GB, 2400 MHz | GeForce GTX 1060 6 GB |

Tabla 1: Especificaciones del dispositivo utilizado para la evaluación.

Además de emplear este portátil, se utilizó una herramienta online denominada Codeanywhere [18], la cual nos permite tener acceso a un IDE en la nube con unas prestaciones más altas y que no se ve afectado por procesos que se están ejecutando internamente. Esto es gracias a que, en vez de utilizar máquinas virtuales, Codeanywhere emplea contenedores. Con ellos, solo se virtualiza el sistema operativo en lugar del ordenador subyacente a la máquina virtual, por lo que son más livianos y requieren menos espacio de memoria. De esta forma, las características del contenedor "básico" gratuito que ha sido empleado para desarrollar parte de este proyecto aparecen en la Tabla 2. Sin embargo, cabe destacar que en la página web de Codeanywhere no está disponible toda la información de los servidores por motivos comerciales.

| Codeanywhere | | |
|---|---|---|
| Almacenamiento | RAM | Swap RAM |
| 10 GB | 2048 MB | 512 MB |

Tabla 2: Especificaciones disponibles de Codeanywhere.

Por último, se empleó la herramienta Visual Studio 2017 [21] para obtener resultados, al igual que en Atom y Codeanywhere, pero los datos obtenidos fueron muy erráticos, ya que estos eran muy dispares entre ejecuciones, por lo que se prescindió de ellos para que no afectaran a las conclusiones finales. De igual manera, con el porcentaje de uso de CPU obtenido en la herramienta *Generador de perfiles de rendimiento* sobre el rendimiento de CPU, no fue útil para extraer resultados por los mismos motivos, aunque a diferencia del caso anterior, sí que sirvió para reforzar la idea de que la función **lfsr_update** era la que más tiempo consumía.

## 4.2 Herramientas empleadas para la medición

Teniendo en cuenta los resultados que mi tutora, Dña. Jezabel Molina Gil obtuvo en el análisis e implementación del generador SNOW 3G [2], donde la función que más consumió, con diferencia, fue la implementación del LFSR, esperamos que ocurra lo mismo con el generador de SNOW-V, debido a que la estructura mayoritaria del generador es muy parecida en ambos casos.

Es importante recalcar que para el análisis de la eficiencia de las funciones del código de SNOW-V nos hemos centrado en dos mediciones: el tiempo total que cada función ha empleado para ejecutarse y el tiempo de CPU que ha consumido.

Para poder hallar el tiempo total que cada función empleó en ejecución, se hizo uso de dos librerías de C++: Ctime y Chrono. Tras los primeros análisis, se decidió descartar a Ctime debido a que Chrono mostraba resultados directamente en microsegundos y además tardaba medio segundo menos en ejecutar todo el código. Otro aspecto que hay que mencionar es que el código tardaba muy pocos microsegundos en ejecutarse, haciendo muy difícil su comparación. Por todo esto, se tomó la decisión de que el generador creara diez mil secuencias cifrantes en vez de una.

En la Tabla 3, se puede apreciar el tiempo en microsegundos (μs) que ha tardado cada función en ejecutarse en el dispositivo de la Tabla 1, empleando Atom como IDE y con la PowerShell de Windows. Como se esperaba, la función que más tiempo ha consumido ha sido lfsr_update, que es la encargada de calcular el registro de retroalimentación de ambos LFSRs. La función **keyiv_setup** ha tardado un tiempo muy parecido ya que dentro de ella se hace una llamada a la función del LFSR, lo que aumenta drásticamente su tiempo.

| Función | Tiempo (μs) | % |
|---|---|---|
| aes_enc_round | 73.735 | 4,655 |
| mul_x | 0 | 0 |
| mul_x_inv | 0 | 0 |
| permute_sigma | 30.885 | 1,94994 |
| fsm_update | 14.961 | 0,94457 |
| lfsr_update | 727.240 | 45,9147 |
| keystream | 26.983 | 1,70359 |
| keyiv_setup | 710.091 | 44,8319 |
| **Total** | 1.583.895 μs | 100 % |

Tabla 3: Rendimiento de las funciones de SNOW-V en el ordenador portátil.

En la Tabla 4 están los resultados obtenidos tras ejecutar el mismo código pero en el IDE online Codeanywhere. Como se puede apreciar, todas las funciones aceleraron bastante su proceso de ejecución y el tiempo total se redujo aproximadamente en un tercio. Igual que ocurrió en el portátil, la función lfsr_update es la que más tiempo ha consumido con diferencia.

| Función | Tiempo (μs) | % |
|---|---|---|
| aes_enc_round | 2.252 | 0,21025 |
| mul_x | 0 | 0 |
| mul_x_inv | 0 | 0 |
| permute_sigma | 8.760 | 0,817845 |
| fsm_update | 480 | 0,0448135 |
| lfsr_update | 493.530 | 46,0766 |
| keystream | 882 | 0,0823447 |
| keyiv_setup | 565.203 | 52,7681 |
| **Total** | 1.071.107 μs | 100 % |

Tabla 4: Rendimiento de las funciones de SNOW-V en Codeanywhere.

Una vez hallado el tiempo total que cada función empleó en ejecutarse, hay que analizar el tiempo de CPU que ha consumido cada función. Para esta tarea se ha empleado el Generador de perfiles de rendimiento que está implementado en el IDE Visual Studio Code 2022. Al igual que antes, todas las pruebas fueron realizadas bajo las mismas condiciones y empleando el mismo código. Tras ejecutar el código con el Generador de perfiles de rendimiento, se obtuvieron los resultados que aparecen en la Tabla 5.

| Función | Procesos CPU | % |
|---|---|---|
| lfsr_update | 171 | 67,86 |
| aes_enc_round | 34 | 13,49 |
| permute_sigma | 12 | 4,76 |
| Otros | 15 | 5,95 |
| **Total** | 232 procesos | 100 % |

Tabla 5: Tiempo de CPU consumido por las funciones.

Como se puede observar en la Tabla 5, más de dos tercios del tiempo de CPU fue usado por la función lfsr_update, un 13.49 % para el método **aes_enc_round**, encargado de realizar las operaciones de AES, un 4,76 % para **permute_sigma**, que como su propio nombre indica, es la encargada de realizar la permutación sigma, y un 5,95 % para el resto de procesos y métodos, que debido a su bajo porcentaje no eran mostrados por la aplicación. Estos datos fueron obtenidos usando las herramientas *Llamador y destinatario* y *Funciones* de Visual Studio Code 2022 que se puede apreciar de mejor manera en la Figura 3 y en la Figura 4.

Figura 3: Llamador y destinatario del Generador de perfiles de rendimiento.

| | |
|---|---|
| main | 242 (96,03 %) |
| SnowV32::keystream | 236 (93,65 %) |
| SnowV32::lfsr_update | 171 (67,86 %) |
| msvcp140.dll!0x00007ffc5b7f3a2f | 160 (63,49 %) |
| SnowV32::keyiv_setup | 157 (62,30 %) |
| ntdll.dll!0x00007ffc610301a5 | 104 (41,27 %) |
| SnowV32::aes_enc_round | 34 (13,49 %) |
| ntdll.dll!0x00007ffc610301b7 | 12 (4,76 %) |
| SnowV32::permute_sigma | 12 (4,76 %) |
| ntdll.dll!0x00007ffc610301cf | 10 (3,97 %) |

Figura 4: Funciones del Generador de perfiles de rendimiento.

Como se esperaba, la función lfsr_update ha consumido gran parte de los recursos tanto en el tiempo total que cada función empleó para ejecutarse como en el tiempo de CPU que ha consumido cada función. Es por ello que en el Capítulo 5 se van a exponer 5 implementaciones software donde se buscará mejorar la eficiencia de ambos LFSRs para así conseguir reducir el tiempo y los recursos que consume este método.

# Capítulo 5

# Análisis de diferentes técnicas para implementar el LFSR

Debido a que SNOW-V se ejecuta en un entorno virtualizado, la implementación del LFSR tiene que ser obligatoriamente software. El problema viene porque los LFSR se han diseñado tradicionalmente para operar sobre el campo de Galois binario [6], ya que este enfoque es apropiado para implementaciones hardware, como ocurría con los anteriores cifrados de la familia SNOW y no a través de software, como ocurre con SNOW-V.

Esta pobre eficiencia en software es debido a dos inconvenientes importantes. El primero de ellos es que para actualizar el estado de un LFSR en una implementación hardware ocurre simultáneamente en un único ciclo de reloj, pero en una implementación software un procesador tiene que gastar muchos ciclos de reloj para realizar las operaciones de desplazamiento de los registros. Además, si la longitud de los LFSRs excede el tamaño de palabra de los procesadores, estas operaciones requerirán mucho tiempo. En segundo lugar, el LFSR binario proporciona un único bit de salida por pulso de reloj, lo que hace otra vez que las implementaciones por software sean muy ineficientes e implica un claro desperdicio de las capacidades de los procesadores modernos.

Es por todo ello que este apartado estará centrado en analizar varias técnicas existentes para mejorar el tiempo de ejecución y de CPU comparado con el código que ofrecen los autores de SNOW-V. Como se comentó en el apartado anterior, nos vamos a centrar en la función del LFSR denominada lfsr_update, ya que es la que más tiempo y uso de CPU consume.

## 5.1 Implementación Tradicional

Esta implementación Tradicional es la que viene implícita en el código que usaron los autores cuando publicaron el artículo de SNOW-V [3]. En este caso, y debido a que se usan dos bucles for, el orden de eficiencia será cuadrático **O(n$^2$)**, ya que esta implementación software almacena los valores de los registros en una matriz. A diferencia de lo que ocurría en SNOW 3G donde solo había un LFSR, en SNOW-V existen dos, por lo que la salida del LFSRA es almacenada en la variable **u** mientras que la salida del LFSRB es guardada en la variable **v**.

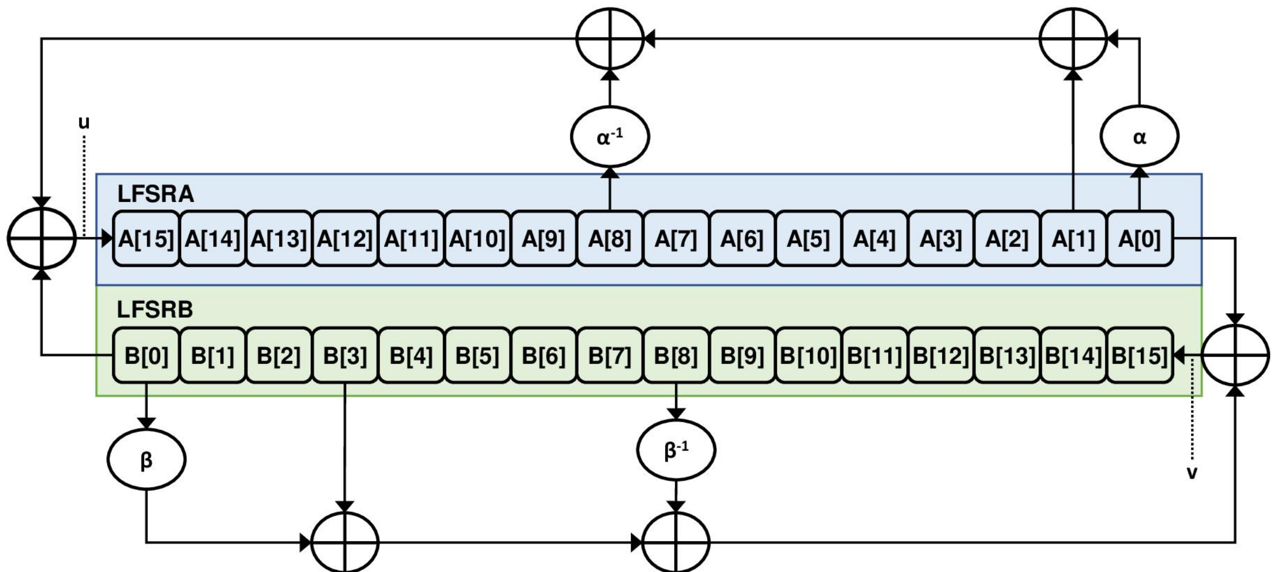


Figura 5: LFSR de la implementación Tradicional.

Como se puede apreciar en la Figura 5, primero se realiza una multiplicación entre el valor que hay en el primer registro del LFSRA, es decir **A[0],** y α. A continuación, se realiza una operación XOR entre el resultado de la multiplicación anterior y el segundo registro del LFSRA, en este caso **A[1]**. El siguiente paso es realizar la división entre el valor que se encuentra en el registro **A[8]** y $\alpha^{-1}$ y realizar una XOR entre este resultado y el del paso anterior. Para finalizar, se realiza una XOR entre el resultado anterior y el valor del primer registro del LFSRB **B[0]**.

Para obtener el valor de v, las operaciones son muy parecidas, solo que se accede a los registros del LFSRB **B[0]**, **B[3]** y **B[8]**, además del registro **A[0]** del LFSRA.

Ahora se realiza el desplazamiento a la derecha de los registros de ambos LFSRs. En este caso, cada LFSR tiene 16 registros, pero en el bucle solo se realiza un desplazamiento desde el primero al penúltimo, ya que el último registro será realimentado por el valor de u para el LFSRA y por v para el LFSRB. El código de la implementación Tradicional es el siguiente:

```c
void lfsr_update(void) {
  for (int i = 0; i < 8; i++) {
    u16 u = mul_x(A[0], 0x990f) ^ A[1] ^ mul_x_inv(A[8], 0xcc87) ^ B[0];
    u16 v = mul_x(B[0], 0xc963) ^ B[3] ^ mul_x_inv(B[8], 0xe4b1) ^ A[0];
    for (int j = 0; j < 15; j++) {
      A[j] = A[j + 1];
      B[j] = B[j + 1];
    }
    A[15] = u;
    B[15] = v;
  }
}
```

## 5.2 Implementación Hardcode

La implementación Hardcode fue la utilizada de forma implícita por los autores para crear el cifrado de flujo SNOW 3G. Esta versión es muy parecida a la implementación Tradicional, con la diferencia de que se elimina el bucle for que desplazaba todos los registros a la derecha para **asignar de forma directa** cada valor al siguiente. De esta forma, lo que se consigue es que el orden de eficiencia sea lineal **O(n)**, en vez de cuadrático como ocurría con la opción Tradicional. Esto puede suponer una clara mejora que será analizada en el siguiente capítulo, aunque no tendría mucho sentido que los autores de SNOW-V hayan implementado una versión menos eficiente que su anterior cifrado.

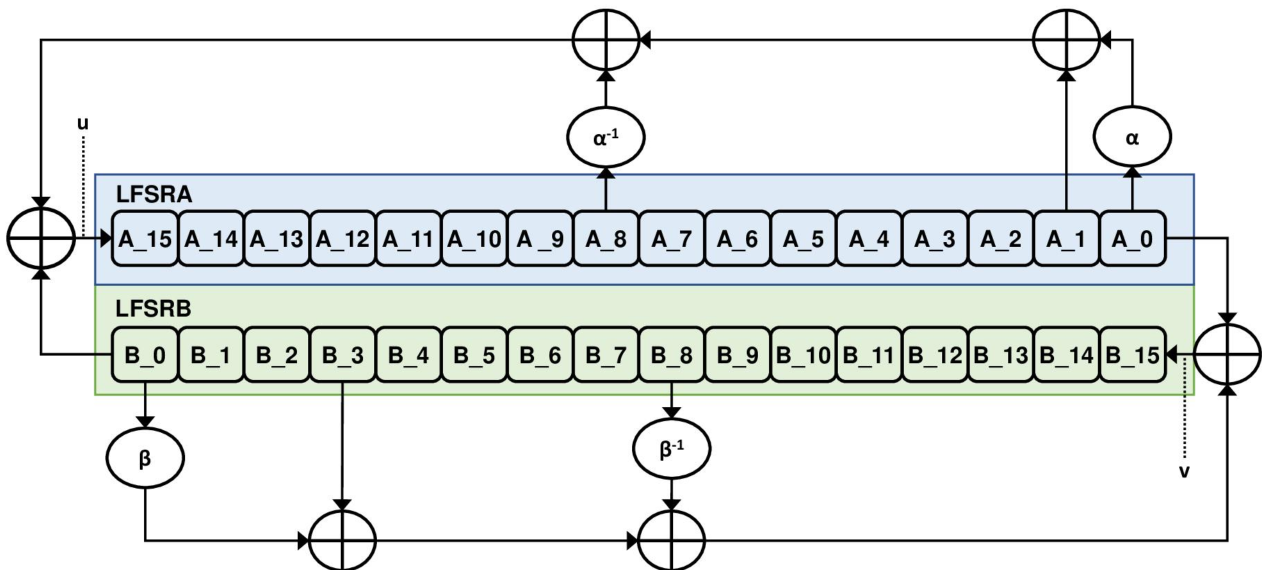


Figura 6: LFSR de la implementación Hardcode.

A la hora de implementar esta versión, hubo que eliminar el bucle for y escribir las asignaciones de los registros una a una hasta un total de 30, 15 para cada LFSR. Esta implementación de software, aunque sea más larga, cabe esperar que aumente la eficiencia por lo comentado en el párrafo anterior.

Hay que tener en cuenta que a partir de esta implementación, hubo que crear el código del LFSR desde prácticamente cero, aunque sí es verdad que ya existían para SNOW 3G, en algunos casos cambia bastante la forma de implementarlo. La función sería la siguiente:

```
void lfsr_update(void) {
  u16 A_0 = 0, A_1 = 1, A_2 = 2, A_3 = 3, A_4 = 4, A_5 = 5, A_6 = 6, A_7 = 7, A_8 = 8,
      A_9 = 9, A_10 = 10, A_11 = 11, A_12 = 12, A_13 = 13, A_14 = 14, A_15 = 15;
  u16 B_0 = 0, B_1 = 1, B_2 = 2, B_3 = 3, B_4 = 4, B_5 = 5, B_6 = 6, B_7 = 7, B_8 = 8,
      B_9 = 9, B_10 = 10, B_11 = 11, B_12 = 12, B_13 = 13, B_14 = 14, B_15 = 15;
```

```
  for (int i = 0; i < 8; i++) {
    u16 u = mul_x(A[A_0], 0x990f) ^ A[A_1] ^ mul_x_inv(A[A_8], 0xcc87) ^ B[B_0];
    u16 v = mul_x(B[B_0], 0xc963) ^ B[B_3] ^ mul_x_inv(B[B_8], 0xe4b1) ^ A[A_0];

    A[A_0] = A[A_1];
    A[A_1] = A[A_2];
    A[A_2] = A[A_3];
    A[A_3] = A[A_4];
    A[A_4] = A[A_5];
    A[A_5] = A[A_6];
    A[A_6] = A[A_7];
    A[A_7] = A[A_8];
    A[A_8] = A[A_9];
    A[A_9] = A[A_10];
    A[A_10] = A[A_11];
    A[A_11] = A[A_12];
    A[A_12] = A[A_13];
    A[A_13] = A[A_14];
    A[A_14] = A[A_15];
    A[A_15] = u;

    B[B_0] = B[B_1];
    B[B_1] = B[B_2];
    B[B_2] = B[B_3];
    B[B_3] = B[B_4];
    B[B_4] = B[B_5];
    B[B_5] = B[B_6];
    B[B_6] = B[B_7];
    B[B_7] = B[B_8];
    B[B_8] = B[B_9];
    B[B_9] = B[B_10];
    B[B_10] = B[B_11];
    B[B_11] = B[B_12];
    B[B_12] = B[B_13];
    B[B_13] = B[B_14];
    B[B_14] = B[B_15];
    B[B_15] = v;
  }
}
```

## 5.3 Implementación de Búfer Circular

La tercera implementación software que se puso en funcionamiento fue Búfer Circular, una técnica comúnmente utilizada en aplicaciones algorítmicas donde hay LFSRs. Usando esta versión, evitamos los desplazamientos de los registros, computacionalmente muy costosos, ya que estos se quedan estáticos y se mueven dos punteros que recorren cada LFSR registro a registro. El nombre que recibe esta implementación viene de que en el instante en el que los punteros llegan al final del LFSR (**búfer**), vuelven al primer registro dando la impresión de ser una circunferencia. Un esquema del funcionamiento del Búfer Circular se puede apreciar en la Figura 7.

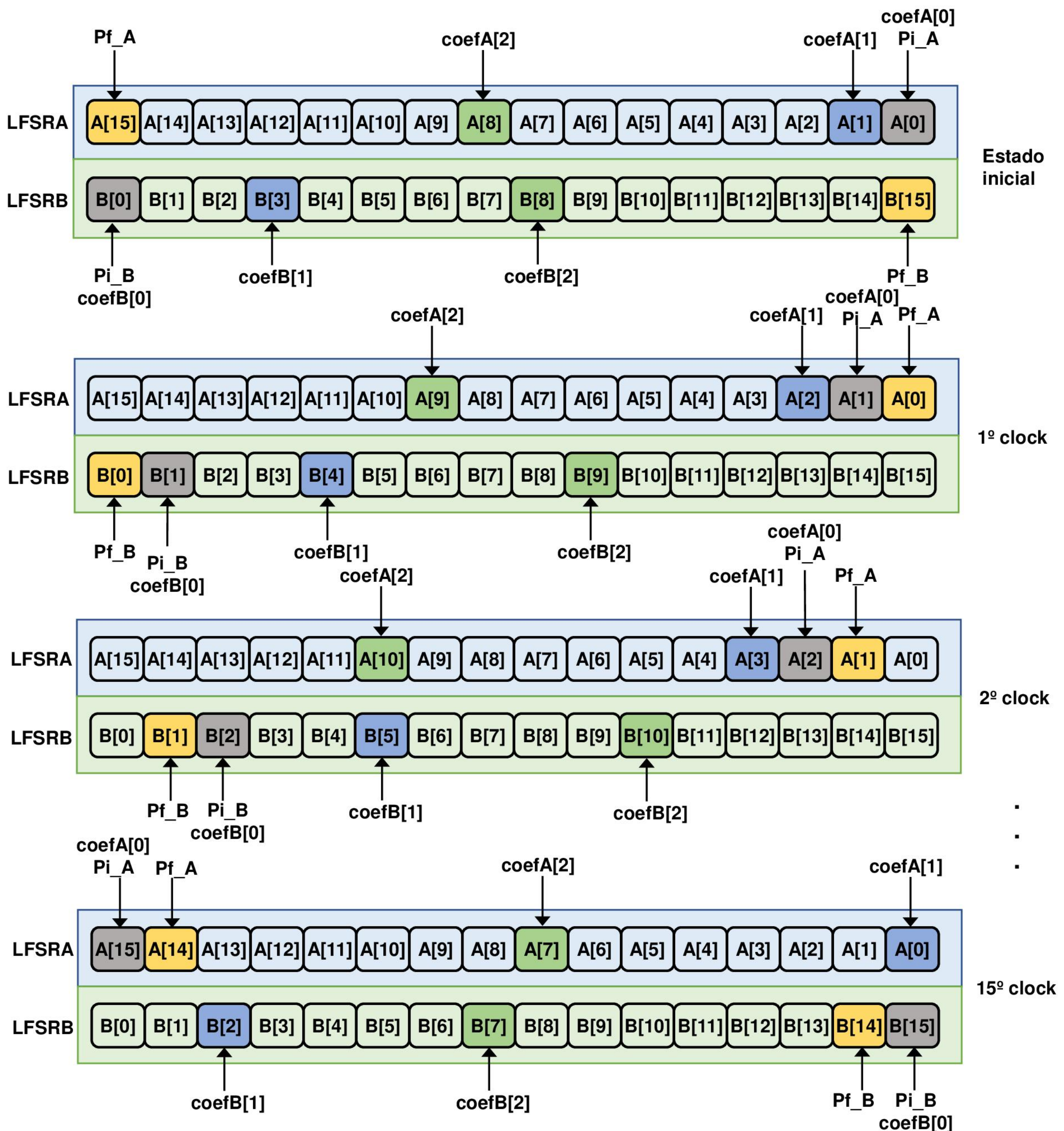


Figura 7: Funcionamiento de la implementación Búfer Circular.

Respecto a la implementación del código para el Búfer Circular, lo primero fue crear las variables de los punteros, denominadas **Pi_A**, para el puntero de inicio del LFSRA y **Pf_A** para el puntero que apunta al final del LFSRA. Extrapolando esto, ocurre lo mismo con **Pi_B** y **Pf_B** para el LFSRB. Hay que remarcar que Pi_A y Pi_B están igualadas a 0, mientras que Pf_A y Pf_B lo están a 15, ya que esos son los valores del primer registro y del último respectivamente. Además, se inicializaron 2 arrays denominados **coefA[3]** y **coefB[3]** de tamaño 3. A cada uno de los elementos del array se les asignó un valor, quedando todo lo mencionado anteriormente de la siguiente manera:

```
//BufferA
Pi_A = 0;
Pf_A = 15;
coefA[0] = 0;
coefA[1] = 1;
coefA[2] = 8;

//BufferB
Pi_B = 0;
Pf_B = 15;
coefB[0] = 0;
coefB[1] = 3;
coefB[2] = 8;
```

Como se puede apreciar, a **coefA[0]** se le asignó el estado 0 del LFSRA, esto es lo mismo que en implementaciones anteriores donde había un **A[0]**. Con esto lo que queremos conseguir es dejar estáticos los registros y simplemente ir moviendo los punteros de cada LFSR para optimizar al máximo la eficiencia de la función. Por último, cabe destacar que el código mostrado anteriormente fue escrito fuera de la función lfsr_update, para no sobreescribir los valores cada vez que se hace una llamada al método.

A diferencia de las anteriores implementaciones, en esta versión hubo que hacer modificaciones más profundas, tanto en la función lfsr_update como en el resto del código. Cada vez que apareció un LFSR, hubo que sumar el puntero inicial a la operación que se realizaba dentro y a todo eso se le aplicó el módulo. Por ejemplo, para el LFSRA donde en anteriores versiones era de la siguiente forma:

```
A[i + 8] = MAKEU16(key[2 * i + 1], key[2 * i]);
```

Ahora tendría que ser así:

```
A[(Pi_A + (i + 8) % 16)] = MAKEU16(key[2 * i + 1], key[2 * i]);
```

Con esto se logró mover el puntero en vez de los estados de los registros y además que no se produjera un desbordamiento al hacer el módulo de 16, el cual es el tamaño máximo del LFSR.

Entrando más en profundidad en la función lfsr_update, la obtención de las variables u y v se realiza de la misma manera que en implementaciones anteriores, solo que accediendo al valor del array en vez de al registro. Otro punto importante es que en cada pasada se aumenta el valor de los 4 punteros que marcan el inicio y el fin y se realiza el módulo de 16. Para finalizar, se procede de la misma forma pero con los valores de los coeficientes y asignando el valor **u** al registro que está siendo apuntado por Pf_A, ya que este es el puntero que marca el último valor. Del mismo modo lo haríamos en el LFSRB con el valor **v** que es apuntado por Pf_B.

```
void lfsr_update(void) {
  for (int i = 0; i < 8; i++) {
      u16 u = mul_x(A[coefA[0]], 0x990f) ^ A[coefA[1]] ^
      mul_x_inv(A[coefA[2]], 0xcc87) ^ B[coefB[0]];

      u16 v = mul_x(B[coefB[0]], 0xc963) ^ B[coefB[1]] ^
      mul_x_inv(B[coefB[2]], 0xe4b1) ^ A[coefA[0]];

      Pi_A = ((++Pi_A) % 16);
      Pf_A = ((++Pf_A) % 16);
      Pi_B = ((++Pi_B) % 16);
      Pf_B = ((++Pf_B) % 16);

      for (int i = 0; i < 15; i++) {
          if (i < coeficientes_LENGTH) {
              coefA[i] = ((++coefA[i]) % 16);
              coefB[i] = ((++coefB[i]) % 16);
          }
      }
    A[Pf_A] = u;
    B[Pf_B] = v;
  }
}
```

# 5.4 Implementación de Ventanas Deslizantes

La implementación de Ventanas Deslizantes usa un método que recibe el mismo nombre para desplazarse por el LFSR mediante el uso de una “ventana”. Esta versión consiste en un búfer el doble de largo, es decir, en duplicar el contenido de ambos LFSRs, pasando de 16 registros en cada uno a 32. La clave consiste en que cuando se calcula un nuevo valor para incluirlo en el LFSR, se escribe en dos posiciones del registro al mismo tiempo, en la primera posición de la ventana y en la posición indicada por el puntero, estas posiciones están representadas en la Figura 8 de color gris.

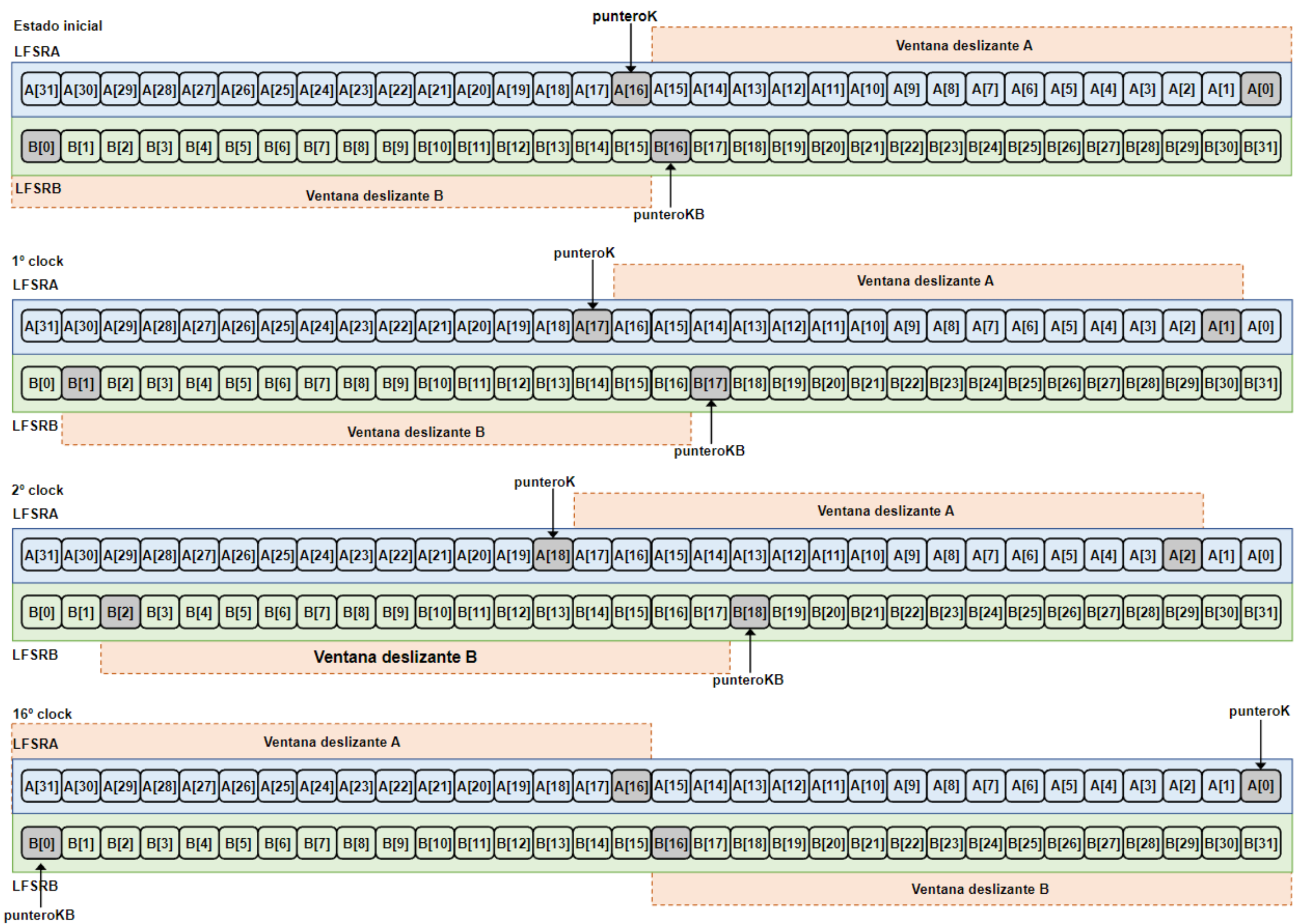


Figura 8: Funcionamiento de la implementación de Ventanas Deslizantes.

Para llevar esto a cabo, se implementa un puntero diferente para cada LFSR, denominado **punteroK** para el LFSRA y **punteroKB** para el LFSRB. Ambos punteros son inicializados en el estado 16, es decir, en el primero de la segunda mitad, indicando así la posición final del registro para cada iteración. Cuando los punteros llegan al final de sus respectivos LFSRs, vuelven al estado 16 de manera cíclica, de forma muy parecida a lo que ocurría en el Búfer Circular. Sin embargo, una ventaja que tiene esta implementación comparada con la de Búfer Circular es que no se realiza ninguna operación módulo, la cual suele tener un coste computacional alto. Además, posee otra ventaja que comparte con el modelo Hardcode al no poseer dos bucles, por lo que su orden de eficiencia será lineal **O(n)**. Además, a diferencia del método de Búfer Circular, el enfoque de ventana deslizante permite que el registro de desplazamiento sea arbitrariamente largo y aun así se dirija de manera eficiente. Todo lo anterior sumado a que su implementación no requiere ampliar el código lo hace un buen candidato para mejorar la eficiencia de los LFSRs.

```
for (int i = 0; i < 8; i++) {
  u32 u = mul_x(A[punteroK - (LFSR_LENGHT + 1)], 0x990f) ^
          A[(punteroK - (LFSR_LENGHT + 1)) + 1] ^
          mul_x_inv(A[(punteroK - (LFSR_LENGHT + 1)) + 8], 0xcc87) ^
          B[punteroKB - (LFSR_LENGHT + 1)];
  u32 v = mul_x(B[punteroKB - (LFSR_LENGHT + 1)], 0xc963) ^
          B[(punteroKB - (LFSR_LENGHT + 1)) + 3] ^
          mul_x_inv(B[(punteroKB - (LFSR_LENGHT + 1)) + 8], 0xe4b1) ^
          A[punteroK - (LFSR_LENGHT + 1)];

  A[punteroK - (LFSR_LENGHT + 1)] = A[punteroK] = u;
  B[punteroKB - (LFSR_LENGHT + 1)] = B[punteroKB] = v;

  if((++punteroK) == (LFSR_LENGHT2 + 2))
    punteroK = (LFSR_LENGHT + 1);

  if((++punteroKB) == (LFSR_LENGHT2 + 2))
    punteroKB = (LFSR_LENGHT + 1);
}
```

## 5.5 Implementación Desdoblamiento de Bucles

La última implementación es Desdoblamiento de Bucles, siendo esta una optimización muy conocida para mejorar el rendimiento de ejecución de un programa. El objetivo principal de esta versión es reducir el número de iteraciones aumentando el cuerpo del bucle for.

Este programa se encuentra en un punto intermedio entre la implementación Tradicional y la Hardcode, ya que mantiene el bucle for obteniendo un orden de eficiencia cuadrático **O(n²)**, pero reduciendo el número de iteraciones de 15 a tan solo 3, ya que el índice aumenta de 5 en 5. Para conseguir esto, en cada iteración del bucle se desplazan 5 estados por cada LFSR, mejorando así la velocidad a cambio de un mayor tamaño del código. Como se puede apreciar en el código inferior, esta implementación es muy parecida a la Tradicional (Figura 5).

```
void lfsr_update(void) {
  for (int i = 0; i < 8; i++) {
    u16 u = mul_x(A[0], 0x990f) ^ A[1] ^ mul_x_inv(A[8], 0xcc87) ^ B[0];
    u16 v = mul_x(B[0], 0xc963) ^ B[3] ^ mul_x_inv(B[8], 0xe4b1) ^ A[0];

    for (int j = 0; j < 15; j = j+5) {
      A[j] = A[j + 1];
      A[j+1] = A[j + 2];
      A[j+2] = A[j + 3];
      A[j+3] = A[j + 4];
      A[j+4] = A[j + 5];
      B[j] = B[j + 1];
      B[j+1] = B[j + 2];
      B[j+2] = B[j + 3];
      B[j+3] = B[j + 4];
      B[j+4] = B[j + 5];
    }
    A[15] = u;
    B[15] = v;
  }
}
```

# Capítulo 6

# Análisis de resultados

Una vez explicadas las diferentes implementaciones software del LFSR que van a ser analizadas, se realizará una comparación entre todas para concluir si la versión Tradicional, implementada en el cifrado de flujo SNOW-V, es la más óptima o se puede mejorar la eficiencia del algoritmo. Para llevar esto a cabo, se han utilizado las herramientas Atom con el dispositivo Asus presentado en la Tabla 1 y el IDE online Codeanywhere. De esta forma se abarcará tanto dispositivos de prestaciones medias como altas para analizar posteriormente sus diferencias y comprobar cómo se comporta el cifrado en diferentes entornos.

El procedimiento para extraer los datos es el mismo que se menciona en el Capítulo 4. Además, para tener unas cifras más precisas, cada implementación ha sido ejecutada 30 veces, almacenando en una tabla todos los tiempos que tardó la función lfsr_update en ejecutarse y realizando la media de estos valores. A continuación, se guardan los datos de la ejecución que más se acerca a la media, siendo este valor el que se va a usar de referencia para saber la mejora de eficiencia respecto al tiempo total de ejecución entre todas las implementaciones. El código con dichas modificaciones se encuentra en el repositorio de GitHub creado para ello [22].

## 6.1 Análisis de resultados con el dispositivo Asus

Los valores obtenidos al ejecutar cada implementación 30 veces con el dispositivo Asus se encuentran en la Tabla 6, donde se resalta en verde la ejecución que más se acerca a los valores promedio y cuyos datos han sido usados para rellenar la Tabla 7, donde se pueden apreciar los tiempos de todas las funciones para cada implementación y el tiempo total.

Los datos de estas tablas muestran que la implementación Tradicional no es la más eficiente ni en la suma total de todas las funciones ni si tenemos solamente en cuenta el método lfsr_update, sino que es la versión de Ventanas Deslizantes la que presenta los mejores resultados, siendo este último un 8,38 % más rápido que el método Tradicional implementado en el cifrado de SNOW-V. Además, representa un 10,07 % de mejora respecto a la implementación de Búfer Circular, siendo este el mayor aumento de rendimiento, posiblemente debido al uso de aritmética modular para actualizar los índices del LFSR, lo que no resulta muy eficiente en una implementación software de un LFSR. En cuanto a la implementación Hardcode, que es la versión implementada en SNOW 3G, se ha alcanzado un 5,89 % de mejora, siendo por tanto más eficiente también que el método tradicional, dejando bastantes dudas del motivo por el que los autores decidieron hacer este cambio en su algoritmo. Por último, la implementación más eficiente después de la de Ventanas Deslizantes es la de Desdoblamiento de Bucles, representando un 5,28 % de mejora respecto a este. Estos datos se presentan en la Tabla 8.

Si en vez de mirar el porcentaje de mejora del tiempo total de Ventanas Deslizantes respecto al resto de implementaciones, comparamos únicamente el tiempo de la función lfsr_update, los resultados son muy similares. Extrayendo los datos de la Tabla 9, la implementación de Ventanas Deslizantes supone un 8,36 % de mejora respecto al Tradicional, un 5,62 % respecto al Hardcode, un 9,07 % respecto al Búfer Circular y un 4,66 % respecto al Desdoblamiento de Bucles.

| | Tradicional | Hardcode | Búfer Circular | Ventanas Deslizantes | Desdoblamiento de Bucles |
|---|---|---|---|---|---|
| **1** | 742.023 | 689.127 | 752.927 | 669.204 | 703.208 |
| **2** | 696.138 | 717.069 | 728.135 | 653.354 | 710.159 |
| **3** | 749.117 | 730.189 | 746.186 | 670.386 | 699.194 |
| **4** | 724.067 | 710.015 | 722.213 | 709.011 | 707.101 |
| **5** | 728.937 | 706.153 | 716.985 | 685.062 | 672.232 |
| **6** | 695.124 | 686.256 | 775.886 | 681.283 | 730.136 |
| **7** | 713.022 | 696.144 | 738.013 | 683.177 | 680.181 |
| **8** | 710.133 | 705.148 | 732.966 | 658.211 | 689.027 |
| **9** | 745.140 | 781.911 | 740.999 | 683.238 | 696.208 |
| **10** | 748.160 | 727.002 | 729.207 | 658.206 | 715.101 |
| **11** | 750.890 | 710.165 | 773.976 | 663.338 | 706.084 |
| **12** | 718.244 | 714.886 | 733.087 | 665.246 | 706.082 |
| **13** | 724.958 | 718.173 | 755.961 | 664.116 | 660.164 |
| **14** | 700.319 | 720.040 | 761.039 | 664.137 | 691.073 |
| **15** | 756.866 | 706.021 | 695.172 | 666.270 | 715142 |
| **16** | 726.163 | 703.960 | 731.839 | 688.270 | 728.978 |
| **17** | 700.060 | 701.070 | 729.052 | 651.295 | 710.207 |
| **18** | 743.123 | 710.054 | 712.815 | 687.220 | 701.211 |
| **19** | 735.980 | 689.121 | 744.050 | 702.837 | 731.175 |
| **20** | 702.225 | 697.171 | 747.090 | 671.278 | 693.193 |
| **21** | 767.062 | 693.210 | 729.069 | 683.208 | 726.975 |
| **22** | 731.966 | 726.087 | 737.987 | 690.993 | 686.064 |
| **23** | 724.213 | 704.088 | 744.097 | 672.433 | 700.256 |
| **24** | 706.039 | 713.990 | 728.070 | 686.075 | 723.117 |
| **25** | 732.997 | 714.125 | 742.967 | 668.255 | 724.022 |
| **26** | 699.070 | 754.006 | 742.932 | 640.215 | 695.091 |
| **27** | 743.090 | 719.131 | 775.986 | 663.165 | 700.131 |
| **28** | 732.960 | 703.893 | 725.900 | 645.235 | 734.198 |
| **29** | 727.240 | 690.110 | 726.072 | 663.214 | 708.187 |
| **30** | 733.022 | 715.233 | 743.768 | 665.201 | 700.182 |
| **Media** | 726.944,93 | 711.784,93 | 738.814,87 | 671.771,1 | 704.858,17 |

Tabla 6: Tiempos de ejecución de lfsr_update empleando el dispositivo Asus.

| Funciones | Tradicional | Hardcode | Búfer Circular | Ventanas Deslizantes | Desdoblamiento de Bucles |
|---|---|---|---|---|---|
| aes_enc_round | 73.735 | 54.888 | 63.865 | 46.867 | 60.832 |
| mul_x | 0 | 0 | 0 | 0 | 0 |
| mul_x_inv | 0 | 0 | 0 | 0 | 0 |
| permute_sigma | 30.885 | 36.853 | 21.012 | 32.919 | 47.844 |
| fsm_update | 14.961 | 14.928 | 23.928 | 22.952 | 18.961 |
| lfsr_update | 727.240 | 710.054 | 738.013 | 671.278 | 703.208 |
| keystream | 26.983 | 30.978 | 33.997 | 17.953 | 34.880 |
| keyiv_setup | 710.091 | 694.295 | 732.892 | 659.254 | 666.361 |
| **Tiempo total** | 1.583.895 | 1.541.996 | 1.613.707 | 1.451.223 | 1.532.086 |

Tabla 7: Tiempos totales de cada implementación empleando el dispositivo Asus.

| | Mejora del total de Ventanas Deslizantes (%) |
|---|---|
| Tradicional | - 8,38 % |
| Hardcode | - 5,89 % |
| Búfer Circular | - 10,07 % |
| Desdoblamiento de Bucles | - 5,28 % |

Tabla 8: Mejora de eficiencia del total de Ventanas Deslizantes respecto al resto de implementaciones empleando el dispositivo Asus.

| | Mejora del LFSR de Ventanas Deslizantes (%) |
|---|---|
| Tradicional | - 8,36 % |
| Hardcode | - 5,62 % |
| Búfer Circular | - 9,07 % |
| Desdoblamiento de Bucles | - 4,66 % |

Tabla 9: Mejora de eficiencia del LFSR de Ventanas Deslizantes respecto al resto de implementaciones empleando el dispositivo Asus.

## 6.2 Análisis de resultados con Codeanywhere

En este apartado se mostrarán los datos extraídos del IDE online Codeanywhere, que al tener unas prestaciones más altas que el dispositivo Asus, permitirá comparar cómo se comporta el cifrado SNOW-V en distintos sistemas y arquitecturas.

Como se puede ver en la Tabla 10, al igual que sucedió con el portátil Asus, la implementación Tradicional no es el método más eficiente para SNOW-V, sino que de nuevo la implementación de Ventanas Deslizantes es mejor, llegando a ser un 21,41 % más eficiente que la Tradicional. Respecto a las otras modificaciones, la segunda más rápida es el Desdoblamiento de Bucles, siendo un 11,13 % menos eficiente que la de Ventanas Deslizantes. La versión Hardcode sigue siendo más rápida que la Tradicional, pero un 14,66 % menos que la implementación óptima. Para finalizar, la modificación del Búfer Circular vuelve a ser la más costosa, alcanzando el 22,47 % respecto a Ventanas Deslizantes. Estos porcentajes se muestran en la Tabla 12.

En la Tabla 11 se encuentran los valores de las 30 ejecuciones de código para cada implementación en Codeanywhere y analizando la Tabla 13, se puede extraer que la modificación de Ventanas Deslizantes reduce en más de un tercio el tiempo de ejecución de la función lfsr_update con respecto a la versión Tradicional, más concretamente en un 37,15 %. En el resto de implementaciones también se producen grandes diferencias comparadas con la más eficiente siendo de media un 31,9 % más rápida que la Hardcode, un 38,68 % que Búfer Circular y un 26,5 % que Desdoblamiento de Bucles.

| Funciones | Tradicional | Hardcode | Búfer Circular | Ventanas Deslizantes | Desdoblamiento de Bucles |
|---|---|---|---|---|---|
| aes_enc_round | 2.252 | 1.677 | 1.035 | 1.174 | 777 |
| mul_x | 0 | 0 | 0 | 0 | 0 |
| mul_x_inv | 0 | 0 | 0 | 0 | 0 |
| permute_sigma | 8.760 | 752 | 527 | 479 | 516 |
| fsm_update | 480 | 234 | 417 | 247 | 199 |
| lfsr_update | 493.530 | 455.367 | 506.377 | 309.063 | 422.193 |
| keystream | 882 | 427 | 446 | 659 | 246 |
| keyiv_setup | 565.203 | 527.964 | 577.004 | 530.176 | 523.266 |
| **Tiempo total** | 1.071.107 | 986.421 | 1.085.806 | 841.798 | 947.197 |

Tabla 10: Tiempos totales de cada implementación empleando Codeanywhere.

|  | Tradicional | Hardcode | Búfer Circular | Ventanas Deslizantes | Desdoblamiento de Bucles |
|---|---|---|---|---|---|
| 1 | 468.737 | 452.442 | 499.748 | 267.784 | 410.013 |
| 2 | 471.520 | 452.154 | 511.083 | 296.936 | 423.347 |
| 3 | 490.833 | 445.998 | 502.570 | 443.420 | 396.404 |
| 4 | 489.155 | 439.581 | 512.608 | 330.619 | 494.674 |
| 5 | 493.530 | 442.918 | 497.094 | 259.008 | 410.179 |
| 6 | 498.024 | 444.825 | 506.377 | 270.512 | 393.131 |
| 7 | 471.102 | 453.435 | 505.486 | 265.607 | 408.986 |
| 8 | 480.131 | 486.738 | 497.764 | 264.491 | 389.512 |
| 9 | 483.928 | 472.834 | 497.825 | 289.522 | 401.774 |
| 10 | 480.947 | 443.297 | 505.240 | 359.514 | 420.606 |
| 11 | 505.295 | 432.931 | 495.444 | 309.063 | 394.266 |
| 12 | 504.428 | 453.861 | 495.032 | 366.752 | 446.270 |
| 13 | 496.235 | 456.277 | 516.948 | 315.984 | 432.670 |
| 14 | 487.487 | 437.199 | 498.963 | 358.072 | 436.292 |
| 15 | 519.204 | 456.403 | 524.456 | 381.855 | 419.843 |
| 16 | 480.409 | 453.685 | 523.960 | 328.033 | 422.193 |
| 17 | 542.821 | 452.121 | 511.730 | 348.224 | 426.590 |
| 18 | 490.833 | 457.296 | 515.015 | 265.141 | 432.540 |
| 19 | 511.490 | 446.032 | 521.539 | 273.897 | 427.103 |
| 20 | 524.216 | 452.563 | 493.693 | 269.586 | 456.916 |
| 21 | 502.598 | 477.900 | 518.631 | 333.739 | 484.485 |
| 22 | 478.413 | 453.927 | 522.172 | 450.183 | 430.126 |
| 23 | 478.186 | 446.657 | 499.275 | 294.276 | 435.943 |
| 24 | 529.754 | 455.367 | 507.938 | 275.847 | 393.804 |
| 25 | 473.499 | 493.224 | 497.439 | 275.271 | 430.825 |
| 26 | 482.179 | 495.391 | 506.543 | 324.208 | 425.944 |
| 27 | 484.756 | 454.552 | 498.409 | 276.108 | 388.154 |
| 28 | 493.824 | 436.786 | 501.108 | 268.043 | 400.935 |
| 29 | 469.018 | 456.517 | 501.769 | 271.964 | 398.417 |
| 30 | 528.259 | 467.738 | 494.160 | 275.184 | 432.686 |
| Media | 493.693,7 | 455.688,3 | 506.000,63 | 310.294,76 | 422.154,26 |

Tabla 11: Tiempos de ejecución de lfsr_update empleando Codeanywhere.

| | Mejora del total de Ventanas Deslizantes (%) |
|---|---|
| Tradicional | - 21,41 % |
| Hardcode | - 14,66 % |
| Búfer Circular | - 22,47 % |
| Desdoblamiento de Bucles | - 11,13 % |

Tabla 12: Mejora de eficiencia del total de Ventanas Deslizantes respecto al resto de implementaciones empleando Codeanywhere.

| | Mejora del LFSR de Ventanas Deslizantes (%) |
|---|---|
| Tradicional | - 37,15 % |
| Hardcode | - 31,9 % |
| Búfer Circular | - 38,68 % |
| Desdoblamiento de Bucles | - 26,5 % |

Tabla 13: Mejora de eficiencia del LFSR de Ventanas Deslizantes respecto al resto de implementaciones empleando Codeanywhere.

# Capítulo 7

# Conclusiones y líneas futuras

El cometido principal de este Trabajo de Fin de Grado ha sido realizar un análisis, tanto desde un punto de vista teórico como práctico de la propuesta del nuevo generador que se va a utilizar para la protección de la integridad y confidencialidad en la generación de telefonía móvil 5G.

De esta manera, se ha realizado una introducción teórica al generador de SNOW-V y se han propuesto y explicado posibles mejoras de implementación para la función que más tiempo consume, las cuales se han implementado en dos plataformas con características distintas, pudiendo obtener resultados en ambas.

En cuanto a las conclusiones, se puede determinar que la implementación Tradicional no es la mejor opción para el cifrado SNOW-V, siendo la técnica de Ventanas Deslizantes la mejor propuesta al ofrecer una mejora respecto al método implementado en el cifrado de entre un 8,38 % y un 21,41 % teniendo en cuenta el tiempo total de ejecución, siendo el primer valor en un portátil de gama media y el último en un IDE online. Además, si miramos los resultados obtenidos analizando únicamente la función del LFSR, la mejora de eficiencia es mayor en el IDE online, llegando a reducir el tiempo de ejecución un 37,15 % con respecto a la técnica Tradicional.

Con todo esto, queda claro que una optimización del software en la telefonía 5G es de vital importancia, ya que la seguridad criptográfica tiene que evolucionar a sistemas más seguros sin permitir que esta se convierta en un cuello de botella para la alta velocidad que supone esta nueva tecnología. En este caso, la implementación de Ventanas Deslizantes reduce de media entre un décimo y un quinto la velocidad total del cifrado SNOW-V sin aumentar el tamaño del código o su complejidad. Esta técnica podría ser apropiada en futuras versiones de la familia SNOW, por lo que sería conveniente que se tuviera en cuenta por los autores.

Como líneas futuras de investigación, se podrían analizar otras implementaciones del LFSR que fueran más eficientes en versión software, aunque las que son más conocidas hoy en día ya han sido implementadas en este Trabajo de Fin de Grado. Además, se podría estudiar si el uso de tablas precalculadas para la multiplicación puede suponer una mejora a costa de aumentar el peso del algoritmo criptográfico, ya que, al ahorrarse calcular constantemente la multiplicación y la división, supuso una mejora en el cifrado SNOW 3G. Otra posible línea de investigación sería analizar por qué los autores decidieron reducir el tamaño de los estados de los registros del LFSR de 32 a 16 bits, ya que lo lógico sería ir aumentando el tamaño de los mismos. Se podría estudiar si se produce alguna mejora al aumentar estos registros a 32 bits como ocurría en SNOW 3G o incluso a 64 bits, ya que la gran mayoría de los ordenadores actuales tienen este tamaño de palabra. Por último, sería interesante analizar e implementar este cifrado en un dispositivo móvil, ya que realmente está pensado para ser utilizado en la telefonía 5G y no en ordenadores.

# Capítulo 8

# Summary and conclusions

The main objective of this Final Degree Project has been to carry out an analysis, from a theoretical and practical point of view, of the proposal of the new generator that will be used for the protection of integrity and confidentiality in the generation of mobile telephony 5G.

In this way, a theoretical introduction to the SNOW-V generator has been made and possible implementation improvements have been proposed and explained for the function that most time consumes, which has been implemented on two platforms with different characteristics, being able to obtain results of both.

Regarding the conclusions, it can be established that the Traditional implementation is not the best option for SNOW-V cipher, being the Sliding Windows technique the best proposal, offering an improvement with respect to the method implemented in the encryption of between 8,38 % and 21,41 % taking into account the total execution time, being the first value in a mid-range laptop and the last in an online IDE. In addition, if we look at the results obtained by analyzing only the LFSR function, the efficiency improvement is greater in the online IDE, reducing the execution time by 37,15 % compared to the Traditional technique.

According to this, it is clear that software optimization in 5G telephony is of vital importance, since cryptographic security has to evolve to more secure systems without allowing it to become a bottleneck for the high speed that this new technology entails. In this case, the implementation of Sliding Windows reduces on average between one tenth and one fifth the total speed of the SNOW-V cipher without increasing the size of the code or its complexity. This technique could be appropriate in future versions of the SNOW family, so it would be convenient for the authors to take it into account.

As future lines of research, other implementations of the LFSR that were more efficient in software version could be analyzed, although the ones that are best known today have already been implemented in this Final Degree Project. In addition, it could be studied whether the use of precalculated tables for multiplication can represent an improvement at the cost of increasing the weight of the cryptographic algorithm, since, by saving the constant calculation of multiplication and division, that meant an improvement in the SNOW 3G cipher. Another possible line of investigation would be to analyze why the authors decided to reduce the size of the LFSR registers from 32 to 16 bits, since it would be more logical to increase their size. It could be studied if there is any improvement when these registers are increased to 32 bits as it occurred in SNOW 3G or even to 64 bits, since the majority of current computers have this word size. Finally, it would be interesting to analyze and implement this encryption on a mobile device, since it is really designed to be used in 5G telephony and not in computers.

# Capítulo 9

# Presupuesto

En este capítulo se mostrará un desglose del presupuesto total del presente Trabajo de Fin de Grado. En total, se han dedicado 335 horas costando cada una de ellas 15 €, lo que hace un total de 5025 €. Además, la versión premium de Codeanywhere cuesta 40 € al mes y como se ha utilizado 2 meses suman 80 €. Por otro lado, el dispositivo empleado para la compilación y ejecución de las implementaciones cuesta 1060,27 €. Si sumamos todo esto, el presupuesto total del proyecto es de 6205,27 €.

| Descripción | Cantidad (Horas) | Precio / Hora (€) | Total (€) |
|---|---|---|---|
| Planificación del proyecto | 20 | 15 | 300 |
| Análisis y lectura de documentación | 40 | 15 | 600 |
| Implementación del cifrado | 50 | 15 | 750 |
| Análisis de resultados preliminares | 5 | 15 | 75 |
| Lectura bibliográfica de posibles modificaciones del LFSR | 20 | 15 | 300 |
| Implementación de las modificaciones | 60 | 15 | 900 |
| Lectura de documentación de diferentes entornos e IDEs donde extraer resultados | 10 | 15 | 150 |
| Análisis y recopilación de resultados definitivos | 80 | 15 | 1.200 |
| Redacción de la memoria | 50 | 15 | 750 |
| Dispositivo Asus GL502VM-FY213T | - | - | 1.060,27 |
| Cuenta premium en Codeanywhere | - | - | 120 |
| **Total** | 335 horas | - | 6.205,27 € |

Tabla 14: Presupuesto total del proyecto.

# Bibliografía


[1] Saqlain, J. (2018). IoT and 5G: History evolution and its architecture their compatibility and future. https://core.ac.uk/download/pdf/161424681.pdf

[2] Molina-Gil, J., Caballero-Gil, P., Caballero-Gil, C., Fúster-Sabater, A. (2014). Analysis and Implementation of the SNOW 3G Generator Used in 4G/LTE Systems. In: , *et al.* International Joint Conference SOCO’13-CISIS’13-ICEUTE’13. Advances in Intelligent Systems and Computing, vol 239. Springer, Cham. https://doi.org/10.1007/978-3-319-01854-6_51

[3] Ekdahl, P., Johansson, T., Maximov, A., & Yang, J. (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

[4] Caforio, A., Balli, F. & Banik, S. Melting SNOW-V: improved lightweight architectures. J Cryptogr Eng 12, 53–73 (2022). https://doi.org/10.1007/s13389-020-00251-6

[5] Ericsson. Encryption in virtualized 5G environments. https://www.ericsson.com/en/blog/2020/6/encryption-in-virtualized-5g-environments

[6] Delgado-Mohatar, O., Fúster-Sabater, A., & Sierra, J. M. (2011). Performance evaluation of highly efficient techniques for software implementation of LFSR. Computers & Electrical Engineering, 37(6), 1222-1231. https://doi.org/10.1016/j.compeleceng.2011.04.002

[7] <ctime> (time.h). cplusplus. https://m.cplusplus.com/reference/ctime/

[8] <chrono>. cplusplus. https://m.cplusplus.com/reference/chrono/

[9] Atom. https://atom.io/

[10] Cloud IDE. Codeanywhere https://codeanywhere.com/

[11] Visual Studio 2022. Microsoft. https://visualstudio.microsoft.com/es/vs/

[12] Análisis de uso de CPU sin depurar en el Generador de perfiles de rendimiento (C#, Visual Basic, C++, F#). Microsoft. https://docs.microsoft.com/es-es/visualstudio/profiling/cpu-usage?view=vs-2022

[13] Ekdahl, P., & Johansson, T. (2000, November). SNOW-a new stream cipher. In Proceedings of first open NESSIE workshop, KU-Leuven (pp. 167-168). http://madchat.fr/crypto/hash-lib-algo/snow/snow10.pdf

[14] Ekdahl, P., Johansson, T. (2003). A New Version of the Stream Cipher SNOW. In: Nyberg, K., Heys, H. (eds) Selected Areas in Cryptography. SAC 2002. Lecture Notes in Computer Science, vol 2595. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-36492-7_5

[15] ISO/IEC 18033-4:2011 Information technology — Security techniques — Encryption algorithms — Part 4: Stream ciphers www.iso.org/standard/54532.html

[16] Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Gouget, A., Sibert, H.: Sosemanuk, a fast software-oriented stream cipher. In: eSTREAM, ECRYPT Stream Cipher. ECRYPT-Network of Excellence in Cryptology, Call for stream Cipher Primitives-Phase 2 (2005), http://www.ecrypt.eu.org/stream

[17] ETSI/SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification. https://www.gsma.com/security/wp-content/uploads/2019/05/snow3gspec.pdf

[18] Billet, O., Gilbert, H. (2005). Resistance of SNOW 2.0 Against Algebraic Attacks. In: Menezes, A. (eds) Topics in Cryptology – CT-RSA 2005. CT-RSA 2005. Lecture Notes in Computer Science, vol 3376. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30574-3_3

[19] Brumley, B.B., Hakala, R.M., Nyberg, K., Sovio, S. (2010). Consecutive S-box Lookups: A Timing Attack on SNOW 3G. In: Soriano, M., Qing, S., López, J. (eds) Information and Communications Security. ICICS 2010. Lecture Notes in Computer Science, vol 6476. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-17650-0_13

[20] Srinivas, N. S., & Akramuddin, M. D. (2016, March). FPGA based hardware implementation of AES Rijndael algorithm for Encryption and Decryption. In 2016 international conference on electrical, electronics, and optimization techniques (ICEEOT) (pp. 1769-1776). IEEE. https://doi.org/10.1109/ICEEOT.2016.7754990

[21] Visual Studio 2017. Microsoft. https://docs.microsoft.com/es-ES/previous-versions/visualstudio/visual-studio-2017/ide/?view=vs-2017

[22] Repositorio GitHub con el código de todas las implementaciones. GitHub. https://github.com/racs0men/Analisis_e_implementacion_del_SNOW-V