DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Final

SimdCrypt Library

Organización y Arquitectura del Computador II

Integrante	LU	Correo electrónico
Daniel Nicolás Kundro		

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

El presente informe es llevado a cabo en el contexto de la implementación de una librería de algoritmos de cifrado y de hashing. Más específicamente, estos algoritmos fueron implementados utilizando el modelo de programación SIMD (Single Instruction, Multiple Data). Por lo tanto, a lo largo de este informe se presentarán diversos algoritmos junto con sus pseudocódigos, explicaciones sobre su funcionamiento y sobre qué partes de los mismos pueden ser paralelizadas, con el objetivo de explotar los beneficios de utilizar el modelo SIMD.

Los algoritmos elegidos están categorizados en dos grandes familias.

1.1. Algoritmos de hashing

Estos algoritmos implementan las llamadas funciones de hash, sin embargo, no todas las funciones de hashing son aptas para el uso en tareas relacionadas a la seguridad informática. Por este motivo, si bien se utilizará este termino, se hará refiriéndose exclusivamente a las funciones de hashing criptográficas.

Estas funciones tienen como objetivo poder recibir una cantidad arbitraria de datos y, a partir de estos, generar un resultado de longitud fija de manera determinística. Adicionalmente, deben poseer las siguientes características para poder ser utilizadas de manera segura:

- Es determinística, para un input dado, siempre generará el mismo resultado.
- Se puede computar rápidamente (baja complejidad computacional).
- No es posible recuperar el input de la función a partir del resultado (llamadas funciones de una vía ya que no tienen función inversa). Si bien es posible probar todas las posibles entradas por fuerza bruta hasta encontrar una que se corresponda con el valor que deseamos, esto es inviable computacionalmente dada la gran cantidad de combinaciones a probar.
- Los resultados de la función deben ser muy diferentes para inputs similares. Es decir, cambiar cualquier bit en el input debería cambiar drásticamente el resultado.
- Es inviable computacionalmente encontrar dos inputs diferentes que generen el mismo resultado.

En general, cuando una función de hashing deja de ser aceptada como segura por la comunidad, es debido a que una de estas características es vulnerada. Por ejemplo, recientemente se probó que la función SHA-1 es insegura al poder producir arbitrariamente dos archivos con el mismo valor de hash.

Las funciones de hashing comprendidas en este trabajo son: MD5, SHA-1, SHA-2 (224, 256, 384, 512) y SHA3 (224, 256, 384, 512).

1.2. Algoritmos de cifrado

El objetivo de estos algoritmos consiste en transformar la información que recibe, en un resultado incomprensible y sin un sentido aparente en relación al input original. Sin embargo, a diferencia de las funciones de hashing, los algoritmos de cifrado tienen como finalidad ser reversibles. Luego, estos algoritmos cuentan con dos funciones principales:

- Cifrado: consiste en transformar un input dado en un resultado incomprensible.
- Descifrado: consiste en transformar un input cifrado en su valor original.

El fin de este tipo de algoritmos consiste principalmente en proteger datos y comunicaciones, por lo tanto, es necesario que solo una persona autorizada pueda conocer el contenido real de un dato cifrado. A tal fin, estos algoritmos utilizan una clave que será utilizada tanto en el proceso de cifrado como en el de descifrado, permitiendo que solo aquellos que conozcan la clave puedan acceder a la información.

En general, estos algoritmos funcionan con un bloque de tamaño fijo tanto para su input como para su output. Luego, solo pueden cifrar información del tamaño de ese bloque y, en caso de tener un input de mayor tamaño, se repite el mismo proceso múltiples veces dividiendo el input en bloques de tal tamaño

(existen varias alternativas para realizar este proceso). En el caso del descifrado es un proceso inverso y análogo.

Por el contrario, es común que el tamaño de las claves si pueda ser variable. Por ejemplo, el algoritmo AES permite llaves de 128, 192 y 256 bits.

Los algoritmos de cifrado comprendidos en este trabajo son: AES (128, 192, 256) y Blowfish.

2. Detalles técnicos y decisiones generales de implementación

A continuación se explicarán diferentes decisiones, detalles técnicos y factores limitantes sobre el desarrollo de la librería a nivel general. Las decisiones puntuales sobre cada algoritmo serán desarrolladas en las siguientes secciones.

2.1. Entorno de desarrollo

La librería fue implementada en una computadora con las siguientes características:

- CPU: Intel® Core™ i5-2400
- Memoria: 3.8 GiB
- SO: Ubuntu 16.04 64-bit

Como puede observarse, el procesador representa un factor limitante ya que dispone únicamente de los siguientes sets de instrucciones: SSE4.1, SSE4.2 y AVX. Si bien estas instrucciones alcanzan para cumplir perfectamente el objetivo de este trabajo, en ciertos algoritmos (especialmente los más modernos) es posible lograr un mayor nivel de paralelización mediante el uso de sets de instrucciones más modernos.

Por otra parte, al haber sido desarrollada en un sistema de 64 bits, la librería solo puede ser utilizada en tales sistemas con arquitectura x86-64. Cabe destacar que contar con instrucciones y registros de 64 bits permite lograr una mayor performance en ciertos algoritmos que fueron diseñados para tales sistemas y, a su vez, no representa un problema a la hora de desarrollar algoritmos más antiguos.

2.2. Decisiones de implementación

2.2.1. Criterios de implementación

Con el fin de maximizar la performance de los algoritmos implementados, los mismos fueron desarrollados con los siguientes criterios:

- Todos los algoritmos fueron implementados tratando de aprovechar lo más posible la paralelización de operaciones mediante el uso de instrucciones SIMD contenidas en los sets de instrucciones anteriormente mencionados.
- Se priorizó reducir al máximo la cantidad de lecturas y escrituras a memoria innecesarias, almacenando la mayor cantidad posible de información en registros.
- Se priorizó reducir la cantidad de saltos condicionales y ciclos con el fin de beneficiar el flujo de ejecución (reduciendo las posibles penalidades correspondientes al pipeline). Esto fue realizado, por ejemplo, mediante la técnica de optimización “loop unrolling”.

Si bien se priorizó obtener una buena performance, este no es necesariamente el objetivo principal del presente trabajo. Por ejemplo, al contar con un set de instrucciones muy amplio, existen instrucciones que implementan algunos de los algoritmos comprendidos en este trabajo. Al estar ya incorporadas en el procesador, estas instrucciones son extremadamente rápidas, luego, si se priorizara la performance sería vital utilizar tales funciones, pero eso le quitaría sentido a este trabajo (donde se busca implementar la totalidad del algoritmo). Por lo tanto, dado que el objetivo no consiste únicamente en lograr una buena performance, también se priorizó mantener un buen nivel de declarativid en el código (equilibrando la influencia de este factor con la de los mencionados previamente).

2.2.2. Modos de operación para algoritmos de cifrado

Como se mencionó en la introducción, los algoritmos de cifrado en bloque pueden operar en diferentes modos, siendo algunos más beneficiosos que otros para ciertas tareas. Los más importantes son:

- Electronic Codebook (ECB)

- Cipher Block Chaining (CBC)
- Propagating Cipher Block Chaining (PCBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

Para realizar este trabajo se decidió no utilizar ningún método en particular, es decir, la librería provee una función para cifrar y descifrar un único bloque. Luego, a partir de esta implementación es posible implementar cualquiera de los modos anteriormente mencionados ya que todos funcionan en base al cifrado y descifrado de un bloque.

2.2.3. Código C y código ASM

La mayor parte del código esta compuesto por la sección de los algoritmos que se encarga de realizar los cálculos y el procesamiento. Este código, dado que se busca paralelizar la mayor cantidad de procesamiento, es implementado en Assembler con el fin de tener un mayor control sobre las instrucciones SSEx y los registros.

Por otra parte, las secciones menos específicas de los algoritmos (donde por lo general se llama a funciones auxiliares que realizan todas las tareas necesarias) están implementadas en C y se encargan de lo siguiente:

- Preparar los datos para que estos puedan ser utilizados directamente por la función que realizará los cálculos necesarios. Esta etapa suele consistir en añadir un padding a los datos de entrada para que estos puedan ser divididos en bloques de igual tamaño.
- Llamar a la función que transforma los datos en el resultado deseado (escrita en Assembler).
- En caso de requerir el resultado en formato ASCII, se encarga de transformar el resultado de la etapa anterior a tal formato.

Como se puede observar, todas estas tareas no requieren paralelización, por lo tanto, fueron implementadas en C.

2.2.4. Tipo de librería

Se decidió que la librería implementada sea una librería estática. Se tomó esta decisión ya que los beneficios que provee una librería dinámica no son influyentes en los contextos de uso donde podría utilizarse esta librería. Por ejemplo, el tamaño de la librería es muy pequeño como para generar un impacto significativo en el tamaño final de un ejecutable, por lo cual no sería necesario que la librería esté por fuera de este. A su vez, por el punto anterior, es más conveniente que la librería se encuentre junto con el resto del código dentro del ejecutable para evitar la necesidad de transportarla externamente. Adicionalmente, dado que esta librería fue desarrollada en el contexto de un trabajo, tiene una única versión. Luego, la capacidad de las librerías dinámicas de ser actualizadas fácilmente tampoco sería explotada.

2.2.5. Interface de la librería

Para incrementar la versatilidad de la librería se decidió que cada algoritmo cuente con dos versiones:

- Algoritmos de hashing:
    - Versión ASCII: el algoritmo recibe como input el puntero a memoria donde se encuentran los datos y la longitud de estos. Luego de realizar su función, devuelve como resultado un puntero al hash en formato ASCII.

    - Versión normal: el algoritmo recibe como input el puntero a memoria donde se encuentran los datos y la longitud de estos. Luego de realizar su función, devuelve como resultado un puntero al hash.

- Algoritmos de cifrado:

    - Versión ASCII: el algoritmo recibe como input el puntero a memoria donde se encuentran los datos. Luego de realizar el cifrado, devuelve como resultado un puntero al resultado en formato ASCII. El algoritmo de descifrado recibe un puntero a un string ASCII y, luego de realizar el descifrado, devuelve un puntero al resultado (si el dato original antes de ser cifrado estaba en formato ASCII, el resultado descifrado estará en formato ASCII).
    - Versión normal: el algoritmo recibe como input el puntero a memoria donde se encuentran los datos. Luego de realizar el cifrado, devuelve un puntero al resultado. El algoritmo de descifrado recibe un puntero a los datos y, luego de realizar el descifrado, devuelve un puntero al resultado.

Es importante destacar que los resultados en formato ASCII ocupan más espacio en memoria y, a su vez, no son iguales al resultado real del hash (a nivel bits). Dado que algunas aplicaciones podrían no necesitar un resultado en formato ASCII, se hace una distinción y se proporcionan ambas funciones. Particularmente, la versión ASCII resulta muy útil para hacer testing de los algoritmos.

Ejemplo de la distinción:

- Hash normal en memoria: 0xA1B2C3D4E5F6
- Hash normal en formato ASCII: 0x413142324333443445354636 (hex). “A1B2C3D4E5F6”string ASCII.

Como se puede observar, la versión ASCII ocupa el doble de espacio en memoria (la parte alta y baja de cada byte debe ser separada en un byte individual para cada una y, posteriormente, convertida a formato ASCII según su valor).

3. Algoritmos de hashing

3.1. MD5

3.1.1. Descripción

Se trata de uno de los algoritmos de hashing más famosos. Habiendo sido publicado en abril de 1992, actualmente se encuentra contraindicado para cualquier uso sensible en seguridad, por ejemplo, en hasheo de contraseñas. Esto se debe a la gran cantidad de vulnerabilidades que han sido encontradas en el algoritmo. Algunas de estas son:

- Colisiones de hash: dos entradas diferentes producen el mismo hash como resultado.
- Ataques de preimagen: es posible encontrar el valor original para un determinado valor de hash con una complejidad computacional mucho menor a la que requeriría utilizar fuerza bruta.

  Este algoritmo puede recibir un input de tamaño variable y produce un hash de 128 bits.

El algoritmo trabaja dividiendo el input recibido en "chunks" de 512 bits, sin embargo, no todos los inputs tienen tal tamaño. Por lo tanto, se aplica un padding al input original que consiste en agregar datos con el fin de que el tamaño del input ampliado sea congruente a 512 modulo 0. El padding esta compuesto de la siguiente manera:

- Se agrega un bit 1 al final del input.
- Se agrega la cantidad necesaria de bits 0 hasta que el tamaño sea congruente a 448 módulo 512.
- Se agrega la longitud del input original en bits. Este número será de 64 bits, logrando que la totalidad del input ampliado sea congruente a 0 módulo 512.

El algoritmo cuenta con cuatro variables de 32 bits (double words) llamadas a0, b0, c0 y d0 (en consistencia con el pseudocódigo de la siguiente sección). Estas variables se modifican en cada iteración correspondiente a cada chunk de 512 bits y son las que finalmente, luego de concatenarlas, darán el hash de 128 bits en el último paso. Debido a la modificación de estas variables por cada chunk, se produce el efecto de que el hash final esté ligado a la totalidad del input.

Como se puede observar en el pseudocódigo, el algoritmo cuenta con ciertas variables cuyos valores están fijos y no dependen del input. Estos valores son utilizados en diversas partes del algoritmo y, por lo tanto, suelen ser construidos de forma aleatoria y transparente con el fin de aumentar la seguridad del mismo y de tener la confianza de la comunidad criptográfica (elegidos de forma maliciosa podrían representar un "backdoor" en el algoritmo). Estas variables son:

- a0, b0, c0, d0: son los valores de inicialización del hash. Con el avance del algoritmo estas variables se modificarán hasta formar el hash final de 128 bits.
- S[64]: contiene 64 valores correspondientes a la cantidad de bits que serán rotados por la función $leftRotate$, según la etapa del algoritmo variara cual de los 64 valores será utilizado.
- K[64]: cada posición "i" corresponde con la siguiente función $floor(2^{32} \times abs(sin(i+1)))$. Estos valores son utilizados en el ciclo central del algoritmo, donde se procesa cada chunk de 512 bits.

3.1.2. Pseudocódigo

Algorithm 1 MD5

```
var S[ 0..63]:= 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21
for i from 0 to 63 do:
    var K[i]:= floor(2^32 × abs(sin(i + 1)))
end for
var a0:= 0x67452301, b0:= 0xefcdab89, c0:= 0x98badcfe, d0:= 0x10325476

function HASH(input, length)
    var paddedMessage:= input
    append "1" to paddedMessage
    while ¬(length of paddedMessage (in bits) ≡ 448 (mod 512)) do
        append "0" to paddedMessage
    end while
    append length to paddedMessage
    for each 512-bit chunk of paddedMessage do:
        break chunk into sixteen 32-bit words M[i], 0≤ i ≤ 15
        var A:= a0
        var B:= b0
        var C:= c0
        var D:= d0
        for i from 0 to 63 do:
            var F, G
            if 0 ≤ i ≤ 15 then
                F:= (B and C) or ((not B) and D)
                G:= i
            else if 16 ≤ i ≤ 31 then
                F:= (D and B) or ((not D) and C)
                G:= (5× i + 1) mod 16
            else if 32 ≤ i ≤ 47 then
                F:= B xor C xor D
                G:= (3× i + 5) mod 16
            else48 ≤ i ≤ 63
                F:= C xor (B or (not D))
                G:= (7× i) mod 16
            end if
            F:= F + A + K[i] + M[G]
            A:= D
            D:= C
            C:= B
            B:= B + leftRotate(F, S[i])
        end for
        a0:= a0 + A
        b0:= b0 + B
        c0:= c0 + C
        d0:= d0 + D
    end for
    var hash:= a0 append b0 append c0 append d0
    Return hash
end function
```

3.1.3. Detalles y decisiones de implementación

- Constantes pre-computadas: con el fin de evitar los tiempos de cómputo de las constantes, las mismas se encuentran pre-computadas en la sección de datos del algoritmo. Dado que no se trata de una gran cantidad de información, la relación aumento de performance/costo de memoria resulta muy favorable.
- En la sección del código encargada de calcular F y G según el valor de i, es posible optimizar ligeramente la performance realizando el siguiente reemplazo (Aumenta la linealidad del cómputo y se reduce la cantidad de instrucciones):
  - En If $0 \leq i \leq 15$ reemplazar F:= (B and C) or ((not B) and D) por F:= D xor (B and (C xor D))
  - En If $16 \leq i \leq 31$ reemplazar F:= (D and B) or ((not D) and C) por F:= C xor (D and (B xor C))
- Carga de constantes: La carga de constantes de inicialización, al tratarse de cuatro double words de 32 bits, puede realizarse directamente a un registro xmm con una sola operación. A su vez, las variables A, B, C y D también pueden ser almacenadas en un registro xmm.
- Rotación de variables: Dado el siguiente código,

  1: A:= D
  2: D:= C
  3: C:= B
  4: B:= B + leftRotate(F, S[i])

  El mismo puede ser optimizado utilizando una única instrucción. Al estar las cuatro variables contenidas en un registro xmm es posible realizar un shuffle de las mismas. Posteriormente se inserta el valor de B luego de realizar la suma.
- Suma de variables: Dado el siguiente código,

  1: a0:= a0 + A
  2: b0:= b0 + B
  3: c0:= c0 + C
  4: d0:= d0 + D

  Al estar contenidas A, B, C y D en un registro y a0, b0, c0 y d0 en otro registro, es posible realizar las cuatro sumas con una única instrucción.
- Guardado del hash: al ocupar 128 bits y estar contenido en las variables a0, b0, c0 y d0 el resultado final puede ser guardado en memoria con una única instrucción sse.

3.1.4. Bibliografía y referencias

- Ronald Rivest, *The MD5 Message-Digest Algorithm*, Abril 1992.
  Disponible en: $https://tools.ietf.org/html/rfc1321$.
- Wikipedia, *MD5*.
  Disponible en: $https://en.wikipedia.org/wiki/MD5$.
- Rosettacode, *MD5, Implementation Debug*.
  Disponible en: $https://rosettacode.org/wiki/MD5/Implementation\_Debug$.

3.2. SHA-1

3.2.1. Descripción

Se trata de un algoritmo diseñado por la National Security Agency (NSA) del gobierno de los Estados Unidos. Habiendo sido publicado en 1995, su utilización en tareas relacionadas con la seguridad informática se encuentra contraindicada. Si bien su uso no era recomendado debido a la existencia de algoritmos más modernos y seguros, el mismo fue declarado inseguro cuando un ataque crítico fue realizado con éxito en febrero de 2017 por investigadores de Google y del CWI. Se trata de un ataque por colisiones y el mismo consistió en construir dos archivos PDF arbitrarios con el mismo valor de hash.

Este algoritmo puede recibir un input de tamaño variable y produce un hash de 160 bits.

El algoritmo trabaja dividiendo el input recibido en "chunks" de 512 bits, sin embargo, no todos los inputs tienen tal tamaño. Por lo tanto, se aplica un padding al input original que consiste en agregar datos con el fin de que el tamaño del input ampliado sea congruente a 512 modulo 0. El padding está compuesto de la siguiente manera:

- Se agrega un bit 1 al final del input.
- Se agrega la cantidad necesaria de bits 0 hasta que el tamaño sea congruente a 448 módulo 512.
- Se agrega la longitud del input original en bits. Este número será de 64 bits, logrando que la totalidad del input ampliado sea congruente a 0 módulo 512.

El algoritmo cuenta con cinco variables de 32 bits (double words) llamadas h0, h1, h2, h3 y h4 (en consistencia con el pseudocódigo de la siguiente sección). Estas variables se modifican en cada iteración correspondiente a cada chunk de 512 bits y son las que finalmente, luego de concatenarlas, darán el hash de 160 bits en el último paso.

En la iteración correspondiente a cada chunk de 512 bits, el mismo es dividido en 16 partes de 32 bits y, en base a esas partes, se comienza a "expandir" el chunk original hasta obtener un chunk de 2560 bits (320 bytes). Luego, la ejecución entra a un ciclo que modifica el valor de las variables temporales a, b, c, d y e utilizando cada parte del chunk expandido. Luego, estas variables se suman a las variables h0, h1, h2, h3 y h4 previamente mencionadas. De esta manera, tales variables son modificadas en base a cada chunk, logrando el efecto de que si cambia cualquier bit de cualquier parte del input, también cambie el hash final.

Como se puede observar en el pseudocódigo, el algoritmo cuenta con ciertas variables cuyos valores están fijos y no dependen del input. Estos valores son utilizados en diversas partes del algoritmo y, por lo tanto, suelen ser construidos de forma aleatoria y transparente con el fin de aumentar la seguridad del mismo y de tener la confianza de la comunidad criptográfica (elegidos de forma maliciosa podrían representar un "backdoor" en el algoritmo). Estas variables son:

- h0, h1, h2, h3 y h4: son los valores de inicialización del hash. Con el avance del algoritmo estas variables se modifican hasta formar el hash final de 160 bits. Los valores de h0, h1, h2 y h3 son iguales que los valores de inicialización de MD5.
- k: esta variable varia en base al valor de i dentro del ciclo central. Estas constantes se corresponden con la raiz cuadrada de 2, 3 ,5 y 10 multiplicado por 2^{30}.

Puede observarse una similitud en la estructura de este algoritmo con respecto a la estructura de MD5. Esto puede deberse a que ambos algoritmos son relativamente contemporáneos, siendo SHA1 ligeramente más moderno. A su vez, también es importante destacar que hay ciertas etapas del algoritmo, como por ejemplo la etapa de inicialización y la etapa de padding, que suelen estar presentes en gran parte de los algoritmos de hashing.

3.2.2. Pseudocódigo

Algorithm 2 SHA-1

```
var h0:= 0x67452301, h1:= 0xEFCDAB89, h2:= 0x98BADCFE
var h3:= 0x10325476, h4:= 0xC3D2E1F0

function HASH(input, length)
    var paddedMessage:= input. append "1" to paddedMessage
    while ¬(length of paddedMessage (in bits) ≡ 448 (mod 512)) do
        append "0" to paddedMessage
    end while
    append length to paddedMessage
    for each 512-bit chunk of paddedMessage do:
        break chunk into sixteen 32-bit words W[i], 0≤ i ≤ 15
        for i from 16 to 79 do:
            W[i]:= (W[i-3] xor W[i-8] xor W[i-14] xor W[i-16]) leftRotate 1
        end for
        var a:= h0
        var b:= h1
        var c:= h2
        var d:= h3
        var e:= h4
        for i from 0 to 79 do:
            var f, k
            if 0 ≤ i ≤ 19 then
                f:= (b and c) or ((not b) and d)
                k:= 0x5A827999
            else if 20 ≤ i ≤ 39 then
                f:= b xor c xor d
                k:= 0x6ED9EBA1
            else if 40 ≤ i ≤ 59 then
                f:= (b and c) or (b and d) or (c and d)
                k:= 0x8F1BBCDC
            else60 ≤ i ≤ 79
                f:= b xor c xor d
                k:= 0xCA62C1D6
            end if
            var temp:= (a leftRotate 5) + f + e + k + W[i]
            e:= d
            d:= c
            c:= b leftRotate 30
            b:= a
            a:= temp
        end for
        h0:= h0 + a
        h1:= h1 + b
        h2:= h2 + c
        h3:= h3 + d
        h4:= h4 + e
    end for
    var hash:= h0 append h1 append h2 append h3 append h4
    Return hash
end function
```

3.2.3. Detalles y decisiones de implementación

- Carga de constantes: La carga de constantes de inicialización, al tratarse de cuatro double words de 32 bits, puede realizarse directamente a un registro xmm con una sola operación. A su vez, las variables A, B, C y D también pueden ser almacenadas en un registro xmm. En ambos casos, la constante restante (h4 en la etapa de inicialización y E en la etapa de hashing) se manejan individualmente en un registro de 32 bits.

- Re-asignación de variables: Dado el siguiente código,

```
1: E:= D
2: D:= C
3: C:= B leftRotate 30
4: B:= A
```

  El mismo puede ser optimizado utilizando instrucciones SSE. Inicialmente se copia el valor de D en E. Luego, estar A, B, C y D contenidas en un registro xmm, es posible realizar un shift de las mismas logrando: D=C, C=B, B=A, A=0. Posteriormente se insertan los demás valores, correspondientes a B rotado en bits y a temp en A.

- Expansión del chunk: En esta etapa se busca ampliar el chunk añadiendo nuevos double words construidos a partir de los correspondientes al input original y, a medida que se construyen, a partir de los recién construidos. El código correspondiente a esta etapa es:

```
1: for i from 16 to 79 do:
2:     W[i]:= (W[i-3] xor W[i-8] xor W[i-14] xor W[i-16]) leftRotate 1
3: end for
```

  En este código, cada W[i] es calculado individualmente, luego, se buscará paralelizar este calculo. Dado que cada W[i] ocupa 32 bits, en un registro XMM hay espacio suficiente para cuatro de estas double words. Luego, al calcular de a cuatro W[i] por iteración, se deben ajustar los offsets de las nuevas instrucciones junto con la cantidad de ciclos, obteniendo el siguiente código:

```
1: while i < 16 do:
2:     W[i] := (W[i-3] xor W[i-8] xor W[i-14] xor W[i-16]) leftRotate 1
3:     W[i+1] := (W[i-2] xor W[i-7] xor W[i-13] xor W[i-15]) leftRotate 1
4:     W[i+2] := (W[i-1] xor W[i-6] xor W[i-12] xor W[i-14]) leftRotate 1
5:     W[i+3] := (W[i] xor W[i-5] xor W[i-11] xor W[i-13]) leftRotate 1
6:     i := i+4
7: end while
```

  Sin embargo, hay un problema, el calculo de W[i+3] utiliza W[i] impidiendo la paralelización de ambas operaciones. Para resolver esta situación, puede utilizarse la siguiente propiedad:
  (A xor B) leftRotate C ≡ (A leftRotate C) xor (B leftRotate C) Utilizándola, se reemplaza W[i] por 0 (elemento neutro del xor), se calcula el resto de la operación de forma paralelizada y posteriormente se realiza el xor con W[i] rotado. Se obtiene el siguiente código:

```
1: while i < 16 do:
2:     W[i] := (W[i-3] xor W[i-8] xor W[i-14] xor W[i-16]) leftRotate 1
3:     W[i+1] := (W[i-2] xor W[i-7] xor W[i-13] xor W[i-15]) leftRotate 1
4:     W[i+2] := (W[i-1] xor W[i-6] xor W[i-12] xor W[i-14]) leftRotate 1
5:     W[i+3] := (0 xor W[i-5] xor W[i-11] xor W[i-13]) leftRotate 1
6:
7:     W[i+3] := W[i+3] xor (W[i] leftRotate 1)
8:     i := i+4
9: end while
```

La paralelización de este código es llevada a cabo de la siguiente manera:

- Se las cargan 16 double words W consecutivas (que serán necesarias para construir las nuevas) en cuatro registros XMM.
- Se reacomoda cada uno de los registros XMM mediante shift's SSE para dejar cada W "en posición".
- Se efectúa la operación XOR entre los cuatro registros utilizando la correspondiente instrucción SIMD.
- Se efectúa la operación "leftRotate" en los cuatro W contenidos en el registro XMM simultaneamente mediante instrucciones SIMD.
- Se extrae W[i] y W[i+3] del registro xmm, se rota W[i], se efectúa el XOR entre ellas y se inserta el resultado en el registro XMM sobreescribiendo a W[i+3].
- Finalmente se guardan los cuatro valores W[i]..W[i+3] en memoria mediante una única instrucción SIMD.

- Suma de variables: Dado el siguiente código,

```
1: h0:= h0 + A
2: h1:= h1 + B
3: h2:= h2 + C
4: h3:= h3 + D
5: h4:= h4 + E
```

  Al estar contenidas A, B, C y D en un registro xmm y h0, h1, h2 y h3 en otro registro xmm, es posible realizar las cuatro sumas con una única instrucción. Luego, la suma restante es realizada mediante una suma ordinaria entre registros de 32 bits.

- En la sección de código correspondiente al ciclo central (donde según el valor de i se setea el valor de f y k), al tratarse de operaciones muy lineales e interrelacionadas (donde cada iteración depende de la anterior) no es posible aplicar paralelización mediante instrucciones SIMD.

- Guardado del hash: al ocupar 160 bits y estar contenido en las variables h0, h1, h2, h3 y h4, el resultado final puede ser guardado en memoria mediante una instrucción sse (para h0..h3) y una escritura a memoria simple para h4.

3.2.4. Bibliografía y referencias

- Dean Gaudet, *SHA-1 using SIMD techniques*, Diciembre 2004.
  Disponible en: $http://arctic.org/\ dean/crypto/sha1.html$.

- Wikipedia, *SHA-1*.
  Disponible en: $https://en.wikipedia.org/wiki/SHA-1$.

- Metamorphosite, *How data encryption software creates one way hash files using the SHA-1 hashing algorithm*, Noviembre 2007.
  Disponible en: $http://www.metamorphosite.com/one-way-hash-encryption-sha1-data-software$.

3.3. SHA-2 (familia)

3.3.1. Descripción

Se trata de un algoritmo diseñado por la National Security Agency (NSA) del gobierno de los Estados Unidos. Habiendo sido publicado en 2001 por el NIST (National Institute of Standards and Technology) este algoritmo sigue en vigencia y es ampliamente utilizado. Si bien no se han encontrado vulnerabilidades en el algoritmo, hay quienes recomiendan la utilización de algoritmos más modernos, tanto por seguridad como por la evolución del hardware (los algoritmos modernos están optimizados para sistemas de 64 bits mientras que SHA-2 está optimizado para sistemas de 32 bits).

Esta familia de algoritmos está compuesta por los siguientes:

- SHA-224
- SHA-256
- SHA-384
- SHA-512

Todos estos algoritmos pueden recibir un input de tamaño variable y producen un hash (output) de tamaño fijo según la versión del algoritmo que se utilice, pudiendo ser: 224, 256, 384 o 512 bits respectivamente. La estructura de estos algoritmos es muy similar, la diferencia entre ellos radica principalmente en los valores de inicialización y en la construcción del hash final. Adicionalmente, existen otras diferencias importantes entre SHA-224/SHA-256 y SHA-384/SHA-512. Debido a la similitud de los algoritmos, se presentará SHA-256 y luego se explicaran las diferencias con cada algoritmo en detalle.

El algoritmo trabaja dividiendo el input recibido en “chunks” de 512 bits, sin embargo, no todos los inputs tienen tal tamaño. Por lo tanto, se aplica un padding al input original que consiste en agregar datos con el fin de que el tamaño del input ampliado sea congruente a 512 modulo 0. El padding está compuesto de la siguiente manera:

- Se agrega un bit 1 al final del input.
- Se agrega la cantidad necesaria de bits 0 hasta que el tamaño sea congruente a 448 módulo 512.
- Se agrega la longitud del input original en bits. Este número será de 64 bits, logrando que la totalidad del input ampliado sea congruente a 0 módulo 512.

El algoritmo cuenta con ocho variables de 32 bits (double words) llamadas h0, h1, h2, h3, h4, h5, h6 y h7 (en consistencia con el pseudocódigo de la siguiente sección). Estas variables se modifican en cada iteración correspondiente a cada chunk de 512 bits y son las que finalmente, luego de concatenarlas, darán el hash de 256 bits en el último paso.

En la iteración correspondiente a cada chunk de 512 bits, el mismo es dividido en 16 partes de 32 bits y, en base a esas partes, se comienza a “expandir” el chunk original hasta obtener un chunk de 2048 bits (256 bytes). Luego, la ejecución entra a un ciclo que modifica el valor de las variables temporales a, b, c, d, e, f, g y h utilizando cada parte del chunk expandido. Luego, estas variables se suman a las variables h0..h7 previamente mencionadas. De esta manera, tales variables son modificadas en base a cada chunk, logrando el efecto de que si cambia cualquier bit de cualquier parte del input, también cambie el hash final.

Como se puede observar en el pseudocódigo, el algoritmo cuenta con ciertas variables cuyos valores están fijos y no dependen del input. Estos valores son utilizados en diversas partes del algoritmo y, por lo tanto, suelen ser construidos de forma aleatoria y transparente con el fin de aumentar la seguridad del mismo y de tener la confianza de la comunidad criptográfica (elegidos de forma maliciosa podrían representar un “backdoor” en el algoritmo). Estas variables son:

- h0, h1, h2, h3, h4, h5, h6 y h7: son los valores de inicialización del hash. Con el avance del algoritmo estas variables se modifican hasta formar el hash final de 256 bits. Los valores de estas variables se corresponden con los primeros 32 bits de la parte decimal de la raíz cuadrada de los primeros 8 números primos.

- k[0..63]: estos valores son utilizados en el proceso de hashing. Su valor se corresponde con los primeros 32 bits de la parte decimal de la raíz cúbica de los primeros 64 números primos.

Puede observarse una similitud en la estructura de este algoritmo con respecto a la estructura de SHA-1 y, por lo tanto, a la estructura de MD5 (en menor medida). Los tres algoritmos siguen un "patrón" de funcionamiento similar, siendo SHA-256 el más robusto de los tres. Esta similitud en el funcionamiento fue un factor determinante en la elección del algoritmo a utilizar para SHA-3, donde se buscó otro patrón de funcionamiento, ya que se sospechaba que algunas de las vulnerabilidades que afectaban a MD5 y a SHA-1 podrían afectar a SHA-256 (si bien hasta el momento no fueron encontradas).

Por otra parte, las constantes de inicialización y las utilizadas durante el hasheo en SHA-2 son diferentes a la de MD5 y SHA-1.

3.3.2. Pseudocódigo

Algorithm 3 SHA-256

```
var h0:= 0x6a09e667, h1:= 0xbb67ae85, h2:= 0x3c6ef372, h3:= 0xa54ff53a
var h4:= 0x510e527f, h5:= 0x9b05688c, h6:= 0x1f83d9ab, h7:= 0x5be0cd19
for i from 0 to 63 do:
    K[i]:= primeros 32 bits de la parte decimal de ∛numeroPrimo(i)
end for

function HASH(input, length)
    var paddedMessage:= input. append "1" to paddedMessage
    while ¬(length of paddedMessage (in bits) ≡ 448 (mod 512)) do
        append "0" to paddedMessage
    end while
    append length to paddedMessage
    for each 512-bit chunk of paddedMessage do:
        break chunk into sixteen 32-bit words W[i], 0≤ i ≤ 15
        for i from 16 to 63 do:
            s0:= (W[i-15] rightRotate 7) xor (W[i-15] rightRotate 18) xor (W[i-15] rightShift 3)
            s1:= (W[i-2] rightRotate 17) xor (W[i-2] rightRotate 19) xor (W[i-2] rightShift 10)
            W[i]:= W[i-16] + s0 + W[i-7] + s1
        end for
        var a:= h0, b:= h1, c:= h2, d:= h3
        var e:= h4, f:= h5, g:= h6, h:= h7
        for i from 0 to 63 do:
            S1:= (e rightRotate 6) xor (e rightRotate 11) xor (e rightRotate 25)
            ch:= (e and f) xor ((not e) and g)
            temp1:= h + S1 + ch + k[i] + w[i]
            S0:= (a rightRotate 2) xor (a rightRotate 13) xor (a rightRotate 22)
            maj:= (a and b) xor (a and c) xor (b and c)
            temp2:= S0 + maj

            h:= g
            g:= f
            f:= e
            e:= d + temp1
            d:= c
            c:= b
            b:= a
            a:= temp1 + temp2
        end for
        h0:= h0 + a
        h1:= h1 + b
        h2:= h2 + c
        h3:= h3 + d
        h4:= h4 + e
        h5:= h5 + f
        h6:= h6 + g
        h7:= h7 + h
    end for
    var hash:= h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
    Return hash
end function
```

3.3.3. Detalles y decisiones de implementación

- Constantes precomputadas: Todas las constantes utilizadas en el algoritmo (h0..h7 y K[0..63]) fueron precomputadas y las mismas se encuentran en la sección de datos del programa. De esta manera, se ahorra tiempo de cómputo en la etapa de inicialización del algoritmo, bastando con cargar tales constantes precomputadas.

- Expansión del chunk: En esta etapa se busca ampliar el chunk añadiendo nuevos double words construidos a partir de los correspondientes al input original y, a medida que se construyen, a partir de los recién construidos. El código correspondiente a esta etapa es:

  1: **for** i from 16 to 63 **do**:
  2: s0:= (W[i-15] rightRotate 7) xor (W[i-15] rightRotate 18) xor (W[i-15] rightShift 3)
  3: s1:= (W[i-2] rightRotate 17) xor (W[i-2] rightRotate 19) xor (W[i-2] rightShift 10)
  4: W[i]:= W[i-16] + s0 + W[i-7] + s1
  5: **end for**

  En este código, cada W[i] es calculado individualmente, luego, se buscará paralelizar este calculo. Dado que cada W[i] ocupa 32 bits, en un registro XMM hay espacio suficiente para cuatro de estas double words. Luego, se buscará calcular de a cuatro W[i] por iteración. Para lograr esto se deben calcular los valores correspondientes a los cuatro s0 y s1 necesarios para calcular los cuatro W[i..i+3].

  La paralelización de cada iteración del ciclo que calcula W[i..i+3] es llevada a cabo de la siguiente manera:

  - Se las cargan 16 double words W consecutivas (que serán necesarias para construir las nuevas) en cuatro registros XMM.
  - Se calcula S0 y S1:
    - Se cargan las mismas cuatro W[i] consecutivas a ser utilizadas en tres registros XMM diferentes.
    - A cada uno de los tres registros se les aplica el shift o la rotación correspondiente de forma individual a nivel de double word mediante instrucciones SSE.
    - Se aplica la operación XOR entre los tres registros XMM.
  - En este punto S0 está listo, pero S1 no lo está ya que para su calculo utiliza W[i-2]. Luego, para dos valores estará listo, pero para otros los valores no ya que los mismos aun no fueron calculados.
  - Se cargan las W[i-16] y W[i-7] necesarias para el calculo de W[i] y se realiza la suma (mediante instrucciones SSE).
  - En este punto W[i] y W[i+1] están listos. Luego, falta calcular W[i+2] y W[i+3] (que para ser calculados requerian tener disponible el valor de W[i] y W[i+1]).
  - Se efectúan los pasos anteriormente mencionados y se termina de construir S1 para los valores que faltaban.
  - Se realiza la suma final: W[i-16] + s0 + W[i-7] + s1 mediante una instrucción SSE.
  - Se guarda el resultado en memoria mediante una única instrucción SSE.

- Carga de constantes: La carga de constantes de inicialización, al tratarse de ocho double words de 32 bits, puede realizarse directamente a dos registros xmm con dos operaciones. A su vez, las variables a, b, c, d, e, f, g y h también pueden ser almacenadas en dos registros xmm (a..d en uno y e..h en otro).

- Re-asignación de variables: Dado el siguiente código,

```
1: h:= g
2: g:= f
3: f:= e
4: e:= d + temp1
5: d:= c
6: c:= b
7: b:= a
8: a:= temp1 + temp2
```

  El mismo puede ser optimizado utilizando instrucciones SSE. Inicialmente se guarda el valor de d en un registro de 32 bits para resguardarlo. Posteriormente se hace un shift del registro completo en los registros XMM que contienen a las variables a..d y e..h. Finalmente, se suma el valor resguardado de d a la variable temp1 y, por otro lado, a la variable temp1 con la variable temp2 (estas operaciones son llevadas a cabo en registros comunes). Luego, estos valores son insertados en su posición correspondiente dentro de los registros que contienen a a..h mediante instrucciones SSE.

- Suma de variables: Dado el siguiente código,

```
1: h0:= h0 + a
2: h1:= h1 + b
3: h2:= h2 + c
4: h3:= h3 + d
5: h4:= h4 + e
6: h5:= h5 + f
7: h6:= h6 + g
8: h7:= h7 + h
```

  Al contar con cuatro registros XMM tal que dos contienen a h0..h7 y otros dos contienen a a..h, es posible realizar la suma del código mostrado mediante la utilización de dos instrucciones de suma SSE.

- En la sección de código correspondiente al ciclo central (donde se itera i desde 0 a 63 modificando el valor de las variables a..h), al tratarse de operaciones muy lineales e interrelacionadas (donde cada instrucción depende de la instrucción anterior), no es posible aplicar paralelización mediante instrucciones SIMD. A su vez, cada iteración del ciclo depende de la iteración inmediata anterior, lo cual impide aún más aplicar paralelización en esta sección.

- Guardado del hash: al ocupar 256 bits y estar contenido en las variables h0..h7, el resultado final puede ser guardado en memoria mediante la utilización de dos instrucciones SSE (una para h0..h3 y otra para h4..h7).

3.3.4. SHA-224

Este algoritmo es idéntico a SHA-256. La unica diferencia entre ambos reside en la forma en la que se construye el hash final y en los valores de inicialización de las variables h0..h7.

Los valores de iniciación son:

- h0:= 0xc1059ed8
- h1:= 0x367cd507
- h2:= 0x3070dd17
- h3:= 0xf70e5939
- h4:= 0xffc00b31
- h5:= 0x68581511

- h6:= 0x64f98fa7
- h7:= 0xbefa4fa4

En cuanto a la construcción del hash, el mismo se obtiene a partir de la concatenación de h0..h6, es decir, se ignora h7.

3.3.5. SHA-512

Si bien este algoritmo es estructuralmente idéntico a SHA-256, posee diferencias importantes tanto en el nivel de seguridad como en su funcionamiento.

Una de las principales diferencias es el aumento en la cantidad de bits de las constantes, las variables y el tamaño de palabra utilizado en los cálculos, pasando de 32 bits a 64 bits. Esto facilita su implementación y mejora su performance en sistemas de 64 bits.

Otra diferencia es el tamaño de los chunks en los que se divide al input del algoritmo, pasando a ser de 1024 bits en lugar de 512 bits. Esto también impacta directamente en el padding que se le aplicará al input.

Las principales diferencias en el ciclo principal de hashing son el incremento en la cantidad de iteraciones del ciclo central (de 64 a 80) y el cambio en las constantes utilizadas para los desplazamientos y rotaciones.

Los valores de inicialización son:

- h0:= 0x6a09e667f3bcc908
- h1:= 0xbb67ae8584caa73b
- h2:= 0x3c6ef372fe94f82b
- h3:= 0xa54ff53a5f1d36f1
- h4:= 0x510e527fade682d1
- h5:= 0x9b05688c2b3e6c1f
- h6:= 0x1f83d9abfb41bd6b
- h7:= 0x5be0cd19137e2179

Las constantes K pasan a estar calculadas en base a los primeros 80 números primos y en lugar de tomar 32 bits de la parte decimal de su raíz cúbica, se toman 64 bits.

3.3.6. SHA-384

Este algoritmo es idéntico a SHA-512. La única diferencia entre ambos reside en la forma en la que se construye el hash final y en los valores de inicialización de las variables h0..h7.

Los valores de iniciación son:

- h0:= 0xcbbb9d5dc1059ed8
- h1:= 0x629a292a367cd507
- h2:= 0x9159015a3070dd17
- h3:= 0x152fecd8f70e5939
- h4:= 0x67332667ffc00b31
- h5:= 0x8eb44a8768581511
- h6:= 0xdb0c2e0d64f98fa7
- h7:= 0x47b5481dbefa4fa4

En cuanto a la construcción del hash, el mismo se obtiene a partir de la concatenación de h0..h5, es decir, se ignoran h6 y h7.

3.3.7. Bibliografía y referencias

- Shariful Islam, *Secure Hash Algorithm (SHA-512)*, Abril 2016.
  Disponible en: $https://goo.gl/ERVgrU$.
- Wikipedia, *SHA-2*.
  Disponible en: $https://en.wikipedia.org/wiki/SHA-2$.
- David Rabahy, *Valores y operaciones intermedias para el calculo de SHA-256*, Septiembre 2014.
  Disponible en: $https://goo.gl/4QzU5N$.

3.4. SHA-3 (familia)

3.4.1. Descripción

En el año 2006, el NIST organizó la "NIST hash function competition" con el fin de encontrar un nuevo algoritmo estándar de hasheo. Si bien SHA-2 continúa siendo seguro (ya que no se han encontrado vulnerabilidades críticas en el algoritmo), se previó que podría llegar a ser vulnerable ya que su estructura es similar a la de MD5 y SHA-1 (algoritmos con graves problemas de seguridad). Luego, el fin de esta competencia fue conseguir una alternativa segura que funcione de manera diferente a los algoritmos SHA-2.

El algoritmo Keccak, diseñado por Guido Bertoni, Joan Daemen, Michaël Peeters y Gilles Van Assche, resultó el ganador de la competencia. Por lo tanto, la familia SHA-3 es un "caso particular" del algoritmo original Keccak.

Esta familia de algoritmos está compuesta por los siguientes:

- SHA3-224
- SHA3-256
- SHA3-384
- SHA3-512

Todos estos algoritmos pueden recibir un input de tamaño variable y producen un hash (output) de tamaño fijo según la versión del algoritmo que se utilice, pudiendo ser: 224, 256, 384 o 512 bits respectivamente. La estructura de estos algoritmos es muy similar, la diferencia entre ellos radica principalmente en los valores de inicialización, tamaños de bloques, la etapa de padding y en la construcción del hash final.

A diferencia de los otros algoritmos presentados en este informe, este algoritmo trabaja dividiendo el input recibido en bloques cuyo tamaño varía según la versión del algoritmo que se esté utilizando y, a su vez, aplica el padding de manera diferente. El padding está compuesto de la siguiente manera:

- Se agrega un byte 0x06 al final del input original.
- Se agrega la cantidad necesaria de bits 0 hasta que el tamaño sea congruente a 0 módulo $tamañodelbloque$.
- Se sobreescribe el último byte del mensaje extendido con 0x80.

El algoritmo, en cualquiera de sus variantes, está compuesto por tres funciones principales:

- Keccak[r,c]: Es el cuerpo principal del algoritmo. Esta función realiza la etapa de "absorción", donde a partir del input se setea el arreglo state. r es el bitrate del algoritmo y c es su capacidad. En Keccak se puede utilizar cualquier valor para r y c, sin embargo, los valores a utilizar en SHA3 fueron fijados en la estandarización realizada por el NIST. Los valores a utilizar son los siguientes y siempre suman 1600 entre ellos:

	r	c
SHA3-224	1152	448
SHA3-256	1088	512
SHA3-384	832	768
SHA3-512	576	1024

- Keccak-f[1600](A): Es la función encargada de aplicar 24 rounds (24 veces la función Round) sobre el arreglo state. En Keccak esta función es: Keccak-f[r+c](S) pero, al estar implementando SHA-3, r+c es siempre 1600.
- Round[1600](A,RC): Esta función aplica varios tipos de operaciones sobre el arreglo state, que también puede ser visto como una matriz de tamaño 5×5. RC es una constante que depende de la rotación que está siendo llevada a cabo.

El algoritmo cuenta con dos sets de constantes fijas:

- Constantes RC: son utilizadas en la función Round, donde se hace un xor entre la primer quadword del arreglo state y la constante correspondiente.
- Offsets de rotación: son utilizadas en varios pasos de la función Round.

El algoritmo comienza aplicando el padding correspondiente al input e inicializando el arreglo de state. Luego, divide el input extendido en bloques de tamaño r (bits) y comienza a iterar dichos bloques. En cada iteración se hace un XOR entre parte del arreglo state y el bloque, y posteriormente se aplican las 24 rondas al arreglo state. Finalmente, se extrae del arreglo state el hash final según la versión del algoritmo que se esté utilizando. La propiedad de que el hash final dependa de la totalidad del input es consecuencia del paso donde se aplica el xor entre parte del arreglo state y cada bloque.

3.4.2. Pseudocódigo

Algorithm 4 SHA-3

```
1: function KECCAK[R,C](input)
2:     var paddedMessage:= input
3:     append "1" to paddedMessage
4:     while ¬(length of paddedMessage (in bits) ≡ 0 (mod r)) do
5:         append "0" a paddedMessage
6:     end while
7:     replace last byte of paddedMessage with 0x80
8:     for x, y from 0 to 4 do:
9:         var S[x,y]:= 0
10:    end for
11:    for each r-bit chunk of paddedMessage do:
12:        for x, y from 0 to 4 tal que x+5*y<r/64 do:
13:            S[x,y]:= S[x,y] xor Pi[x+5*y]
14:            S:= Keccak-f[r+c](S)
15:        end for
16:    end for
17:    var hash := copy hash size from S first bits
18:    Return hash
19: end function
20:
21: function KECCAK-F[1600](S)
22:    for i from 0 to 23 do:
23:        S:= Round[1600](S, RC[i])
24:    end for
25:    Return S
26: end function
27:
28: function ROUND[1600](S, RC)
29:    for x from 0 to 4 do:
30:        C[x]:= S[x,0] xor S[x,1] xor S[x,2] xor S[x,3] xor S[x,4]
31:    end for
32:    for x from 0 to 4 do:
33:        D[x]:= C[x-1] xor rot(C[x+1],1)
34:    end for
35:    for x, y from 0 to 4 do:
36:        S[x,y]:= S[x,y] xor D[x]
37:    end for
38:    for x, y from 0 to 4 do:
39:        B[y,2*x+3*y]:= rot(S[x,y], r[x,y])
40:    end for
41:    for x, y from 0 to 4 do:
42:        S[x,y]:= B[x,y] xor ((not B[x+1,y]) and B[x+2,y])
43:    end for
44:
45:    S[0,0]:= S[0,0] xor RC
46:    Return S
47: end function
```

3.4.3. Detalles y decisiones de implementación

- Inicialización del arreglo state: Esta etapa puede realizarse de tres formas:
  - Registros XMM: consiste en setear en 0 un registro XMM y copiar su contenido a lo largo de todo el tamaño del arreglo state mediante 12 instrucciones SIMD.
  - Registros YMM: consiste en setear en 0 un registro YMM y copiar su contenido a lo largo de todo el tamaño del arreglo state mediante 6 instrucciones SIMD (esta opción no fue implementada por limitaciones del procesador).
  - Registros ZMM: consiste en setear en 0 un registro ZMM y copiar su contenido a lo largo de todo el tamaño del arreglo state mediante 3 instrucciones SIMD (esta opción no fue implementada por limitaciones del procesador).

  En ambos casos falta setear en 0 la última posición de memoria y esto puede realizarse con un registro común de 64 bits.
- Constantes precomputadas: Todas las constantes utilizadas en el algoritmo (offsets de rotación y constantes RC) se encuentran precomputadas en la sección de datos del programa. Luego, para su utilización basta con cargar tales constantes desde la memoria.
- La función Keccak-f[1600] no puede ser optimizada mediante SIMD ya que cada iteración depende del resultado de la iteración inmediata anterior.
- En la función Keccak[r,c] se puede optimizar la inicialización del arreglo state y la extracción del hash final. Los cálculos realizados en el ciclo principal no pueden ser optimizados mediante SIMD ya que cada iteración depende de los resultados de la iteración anterior.
- Guardado del hash: El tamaño del hash final varía según que versión se este utilizando (224, 256, 384 o 512). El hash final se extrae del arreglo state, luego, basta con copiar el valor desde tal arreglo.
  - Para 224: Se puede copiar mediante el uso de un registro XMM, uno Rxx(64 bits) y uno Exx (32 bits).
  - Para 256: Se puede copiar mediante el uso de dos registros XMM o un registro YMM.
  - Para 384: Se puede copiar mediante el uso de un registro XMM y un registro YMM.
  - Para 512: Se puede copiar mediante el uso de dos registros YMM o un registro ZMM.
- Funcion Round: Esta función es la que realiza la mayor parte de los cálculos y, por ende, es la que debe estar más optimizada. A continuación se detalla como se optimizo cada ciclo de esta función mediante el uso de instrucciones SIMD:
  - 1: **for** x from 0 to 4 **do**:
    2: C[x]:= S[x,0] xor S[x,1] xor S[x,2] xor S[x,3] xor S[x,4]
    3: **end for**

    C[x] es un arreglo de tamaño 5 donde, en cada posición, se guarda el resultado de hacer XOR entre los elementos de cada columna. Luego, puede ser reescrito de la siguiente forma:
    1: C[0]:= S[0,0] xor S[0,1] xor S[0,2] xor S[0,3] xor S[0,4]
    2: C[1]:= S[1,0] xor S[1,1] xor S[1,2] xor S[1,3] xor S[1,4]
    3: C[2]:= S[2,0] xor S[2,1] xor S[2,2] xor S[2,3] xor S[2,4]
    4: C[3]:= S[3,0] xor S[3,1] xor S[3,2] xor S[3,3] xor S[3,4]
    5: C[4]:= S[4,0] xor S[4,1] xor S[4,2] xor S[4,3] xor S[4,4]

    El objetivo es calcular múltiples valores de C simultáneamente. Esto puede ser llevado a cabo cargando las diferentes filas de la matriz en registros y aplicando la operación xor entre ellos (entre filas). Sin embargo, dado que cada S[x,y] ocupa 64 bits, en un registro XMM solo entran

dos. Por este motivo solo se cargaran 4 valores de cada fila (utilizando 10 registros XMM) y los valores correspondientes a la quinta columna serán manejados por separado en registros comunes. Luego, se calculan en paralelo C[0..3] y C[4] se calcula por separado. Los valores de C[0..3] son guardados en memoria mediante una única instrucción SIMD.

- 1: **for** x from 0 to 4 **do**:
  2: D[x]:= C[x-1] xor rot(C[x+1],1)
  3: **end for**

  El cálculo de esta función puede ser paralelizado utilizando SIMD de la siguiente manera:
  - Calculo de C[x-1]: Si bien estos valores fueron calculados en el paso anterior, en esta función son utilizados con un desplazamiento (por ejemplo, para el calculo de D[0] se utiliza C[4]). Luego, es necesario cargar los valores y rotarlos. La carga puede ser llevada a cabo en dos registros XMM y un registro de propósito general. Para llevar a cabo el desplazamiento basta con cargar los valores en el orden deseado: En el registro de propósito general cargo C[4], en un registro XMM cargo C[0] y C[1] y en el otro registro XMM cargo C[2] y C[3].
  - Calculo de rot(C[x+1],1): Al igual que en el punto anterior para guardar todos los valores necesitaremos dos registros XMM y uno de propósito general. A su vez, también cuenta con un desplazamiento, pero este es inverso al anterior (por ejemplo, para el calculo de D[0] se utiliza C[1]). Luego, se comienza cargando los valores de la siguiente manera: En un registro XMM cargo C[1] y C[2], en el otro cargo C[3] y C[4] y en el de propósito general cargo C[0].
    Luego, mediante instrucciones SSE, se realiza la rotación de bits en cada quadword contenida en los registros XMM. Se aplica el mismo proceso en el registro de propósito general utilizando instrucciones comunes. Por último estos valores se guardan en la memoria correspondiente D.

  Una vez calculados todos los valores previamente mencionados, se hace efectúa un xor entre cada C[x-1] y D[x] (mediante el uso de los registros previamente mencionados).

- 1: **for** x, y from 0 to 4 **do**:
  2: S[x,y]:= S[x,y] xor D[x]
  3: **end for**

  Esta función se encarga de hacer un xor entre cada valor de cada columna i de state, con el valor de D[i]. Luego, con el fin de paralelizar su calculo mediante SIMD, se pueden cargar los primeros cuatro valores de D en dos registros XMM. Posteriormente, se cargan los primeros cuatro valores de una fila de S en dos registros XMM. Se ejecuta la operación XOR entre los dos registros que contienen D y los dos registros que contienen la fila de S y se guarda el resultado en S sobrescribiendo los datos que fueron cargados inicialmente. Este proceso se repite para todas las filas de S (la implementación no utiliza un ciclo con el fin de mejorar la performance). Finalmente, se aplica un proceso similar para el quinto valor de cada fila de S (quinta columna) pero utilizando registros de propósito general. Es decir, se carga el valor de la quinta posición de D y se comienza a cargar la quinta posición de cada fila de S ejecutando el XOR y sobrescribiendo su valor. Estos valores se manejan por separado ya que los cinco valores de cada fila no caben en un registro XMM.

- 1: **for** x, y from 0 to 4 **do**:
  2: S[x,y]:= B[x,y] xor ((not B[x+1,y]) and B[x+2,y])
  3: **end for**

  El cálculo de esta función puede ser optimizado mediante instrucciones SIMD. Esta función

opera por filas, por lo tanto se utilizaran dos registros XMM y uno de propósito general para trabajar con la totalidad de cada fila.

El calculo de cada fila es idéntico (variando los offsets), luego, se explicara únicamente el calculo de una fila (S[0..4,y]):

- Se carga B[0..4] en el set de registros mencionados.
- Se carga B[x+1] (B con desplazamiento) en otro set de registros (XMMx: B[1] y B[2], XMMy: B[3] y B[4], proposito general: B[0])
- Se aplica un not en los registros del paso anterior mediante instrucciones SSE (en caso de los registros XMM).
- Se carga B[x+2] (B con desplazamiento) en otro set de registros (XMMx: B[2] y B[3], XMMy: B[4] y B[0], propósito general: B[1]).
- Se ejecuta el and entre los valores de not(B[x+1]) y los de B[x+2].
- Se ejecuta el xor entre los valores de B[x] y los calculados en el paso anterior
- Se guardan los resultados del paso anterior en state, sobreescribiendo su contenido en las posiciones correspondientes.

3.4.4. Bibliografía y referencias

- Guido Bertoni, Joan Daemen, Michaël Peeters y Gilles Van Assche; *Keccak specifications summary*. Disponible en: $https://keccak.team/keccak_specs_summary.html$.
- Guido Bertoni, Joan Daemen, Michaël Peeters y Gilles Van Assche; *The Keccak reference*, Enero 2011. Disponible en: $https://keccak.team/files/Keccak-reference-3,0.pdf$.
- Christof Paar, Jan Pelzl; *SHA-3 and The Hash Function Keccak*. Disponible en: $https://goo.gl/anJ56s$.
- Wikipedia, *SHA-3*. Disponible en: $https://en.wikipedia.org/wiki/SHA-3$.

4. Algoritmos de cifrado

4.1. BLOWFISH

4.1.1. Descripción

Es un algoritmo de cifrado diseñado por Bruce Schneier y publicado en 1993. Si bien es un algoritmo antiguo y se recomienda la utilización de sus sucesores, las vulnerabilidades que se conocen hasta el momento no son críticas, por lo cual sigue siendo utilizado en la actualidad. A diferencia de la mayoría de los algoritmos similares de esa época, Blowfish fue publicado sin patentar y libre para su uso.

Se trata de un algoritmo muy rápido y de uso general. Adicionalmente, debido a su construcción, es especialmente resistente a ataques de fuerza bruta o de diccionario. Esto se debe a que cada vez que cambia la clave, se debe realizar una inicialización del algoritmo en base a tal clave y la inicialización es más costosa que realizar el cifrado en si. Por lo tanto, se obtiene resistencia a los ataques que cambian de claves constantemente.

Blowfish utiliza un tamaño de bloque de 64 bits, 16 rounds y, a diferencia de otros algoritmos como AES, el tamaño de su clave puede ser variable entre 32 y 448 bits.

Para poder comenzar a funcionar, el algoritmo debe inicializar un conjunto de variables que serán utilizadas durante el proceso de cifrado y descifrado. Esta inicialización se lleva a cabo utilizando la clave recibida como input. De esta manera, el algoritmo trabaja con valores que dependen exclusivamente de la clave. Los pasos para la inicialización son:

- pArray: es un arreglo de subclaves. Este arreglo tiene valores por defecto y, en la etapa de inicialización, se lo modifica utilizando la clave original mediante la operación xor. Posteriormente, se vuelve a modificar P utilizando el algoritmo de cifrado (que utiliza los valores de P del paso anterior).
- sBox[0..3][0..255]: Son cuatro arreglos llamados S-Box (substitution box). Como su nombre lo indica, son utilizados para realizar sustituciones (dadas dos coordenadas x e y, se utiliza el valor de sBox[x][y]). Estos arreglos son seteados mediante el uso del algoritmo de cifrado (que utiliza los valores de P previamente calculados).

Una vez realizados estos pasos, el algoritmo está listo para comenzar a cifrar o descifrar el input correspondiente.

Como se puede observar en el pseudocódigo, el proceso de cifrado y descifrado es muy similar, variando principalmente el orden en el que las operaciones son llevadas a cabo. Ambas funciones utilizan los valores calculados en el proceso de inicialización, operaciones XOR y una función “f” encargada de, a partir de un valor, tomar un valor específico de una s-Box.

Las funciones de cifrado y descifrado reciben como input el bloque de 64 bits dividido en dos partes (L y R) de 32 bits cada una. Luego, se realizan 16 rounds en las cuales se utilizan las operaciones mencionadas previamente sobre L y R. Finalmente, se swapea el valor de L y R, se los concatena (formando nuevamente un bloque de 64 bits) y se lo retorna (el bloque inicial cifrado/descifrado).

4.1.2. Pseudocódigo

Algorithm 5 BLOWFISH

```
1: function ENCRYPT(L, R)
2:     while i < 16 do
3:         L:= L xor P[i]
4:         R:= R xor f(L)
5:         R:= R xor P[i+1]
6:         L:= L xor f(R)
7:         i:= i+2
8:     end while
9:     L:= L xor P[16]
10:    R:= R xor P[17]
11:    Swap(L, R)
12:    Return (append (L, R))
13: end function
14:
15: function DECRYPT(L, R)
16:    i:= 16
17:    while i > 0 do
18:        L:= L xor P[i+1]
19:        R:= R xor f(L)
20:        R:= R xor P[i]
21:        L:= L xor f(R)
22:        i:= i-2
23:    end while
24:    L:= L xor P[1]
25:    R:= R xor P[0]
26:    Swap(L, R)
27:    Return (append (L, R))
28: end function
29:
30: function INICIALIZACIÓN(key)
31:    for i from 0 to 18 do:
32:        P[i]:= P[i] xor key[i % key.length]
33:    end for
34:    L:= 0; R:= 0
35:    for i from 0 to 18, i:= i+2 do:
36:        encrypt(&L, &R)
37:        P[i]:= L; P[i+1]:= R
38:    end for
39:    for i from 0 to 4 do:
40:        for j from 0 to 256, j:= j+2 do:
41:            encrypt(&L, &R)
42:            S[i][j]:= L; S[i][j+1]:= R
43:        end for
44:    end for
45: end function
46:
47: function F(X)
48:    H:= (S[0][X >> 24]) + (S[1][X >> 16 and 0xFF])
49:    return (H and (S[2][X >> 8 and 0xFF])) + (S[3][X and 0xFF])
50: end function
```

4.1.3. Detalles y decisiones de implementación

- Constantes pre-computadas: Tanto las s-Box como el p-Array comienzan con valores que serán utilizados para comenzar con el proceso de inicialización. Estos valores se encuentran pre-computados en la sección de datos del algoritmo. Esto evita la necesidad de tener que calcular estos valores en cada ejecución. Adicionalmente, dado que las constantes son modificadas en cada ejecución, se cuenta con dos espacios de memoria. Uno de solo lectura donde se encuentran los valores iniciales y otro de lectura/escritura donde se realizará el proceso de inicialización. Contar con las constantes precomputadas ahorra tiempo de cómputo en el comienzo del algoritmo, mejorando su performance.

- Carga de constantes: Como se mencionó anteriormente, los valores iniciales de las s-Box y del p-Array son copiados desde un espacio de memoria de solo lectura hacia uno de lectura/escritura. Este proceso es llevado a cabo mediante instrucciones SIMD. Particularmente, se utilizan registros YMM con el fin de poder operar con 256 bits simultáneamente, lo cual reduce ampliamente la cantidad de operaciones y acelera la copia de los valores.

- Función f: Esta función no puede ser optimizada utilizando SIMD ya que realiza únicamente operaciones con registros de 32 bits. Adicionalmente, calcular f para varios valores simultáneamente tampoco es viable ya que en las secciones de código donde es utilizada hay una gran dependencia lineal de operaciones (cada operación de cada iteración depende de su inmediata anterior, impidiendo la paralelización).

- Función $encrypt/decrypt$: Estas funciones no pueden ser optimizadas mediante la utilización de SIMD ya que las mismas están compuestas principalmente por un ciclo cuyas operaciones no pueden ser paralelizadas. Todas las operaciones contenidas en el ciclo dependen de la operación inmediata anterior, por lo tanto, se trata de una ejecución muy dependiente y lineal. Adicionalmente, al utilizar bloques de 64 bits divididos en dos partes, la utilización de registros de 32 bits resulta ideal.

- Inicialización: Durante esta etapa es posible utilizar instrucciones SIMD para optimizar el calculo de los valores del p-Array. Se cuenta con un arreglo que contiene a la clave original repitiendose hasta completar 576 bits. Luego, tanto la clave extendida como el p-Array tienen el mismo tamaño. Por lo tanto, es posible realizar el xor entre la clave y el p-Array directamente mediante cuatro instrucciones SIMD sobre registros XMM, dos instrucciones sobre registros YMM o una instrucción sobre registros ZMM. Finalmente, faltan 64 bits correspondientes a los dos últimos valores del p-Array y de la clave extendida, los cuales son procesados en registros de propósito general.

  El resto del proceso de inicialización está compuesto por dos ciclos donde se itera cifrando valores y guardandolos en el p-Array y en las s-Box. Estos procesos no pueden ser optimizados mediante instrucciones SIMD ya que consisten en hacer llamadas a la función de cifrado y escrituras a memoria de 64 bits. A su vez, las iteraciones en si no pueden ser paralelizadas ya que cada iteración depende del resultado cifrado de la iteración anterior.

4.1.4. Bibliografía y referencias

- Bruce Schneier, *The Blowfish Encryption Algorithm*, 1992.
  Disponible en: $https://www.schneier.com/academic/blowfish/$.

- Bruce Schneier, *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, 1992.
  Disponible en: $https://www.schneier.com/academic/archives/1994/09/description\_of\_a\_new.html$.

- Wikipedia, *Blowfish cipher*.
  Disponible en: $https://en.wikipedia.org/wiki/Blowfish\_(cipher)$.

- Wikipedia, *S-box*.
  Disponible en: $https://en.wikipedia.org/wiki/S-box$.

4.2. AES

4.2.1. Descripción

Es un algoritmo de cifrado diseñado por Vincent Rijmen y Joan Daemen, publicado en 1998. Si bien su nombre original es Rijndael, al ganar la competencia organizada por el National Institute of Standards and Technology (NIST) pasó a ser conocido como Advanced Encryption Standard (AES) (siendo este un subconjunto de Rijndael).

Este algoritmo es sumamente utilizado en la actualidad, siendo incluso adoptado por el gobierno de los Estados Unidos. A pesar de tener 20 años, continúa siendo un algoritmo seguro y las únicas vulnerabilidades encontradas hasta el momento son de carácter teórico (permiten reducir la complejidad computacional de un ataque con respecto a un ataque por fuerza bruta, sin embargo, esta complejidad sigue siendo muy alta y por ende impracticable).

AES utiliza un tamaño de bloque fijo de 128 bits (16 bytes) y puede ser utilizado con tres tamaños de clave diferentes: 128, 192 o 256 bits. Según el tamaño de clave utilizado, también variará la cantidad de rounds del algoritmo: 10, 12 o 14. En cuanto a su estructura, a diferencia de otros algoritmos de cifrado como Blowfish, utiliza una red de sustitución-permutación.

El funcionamiento del algoritmo puede ser resumido en los siguientes pasos:

Cifrado	Descifrado
1. KeyExpansion 2. round inicial: *a*) AddRoundKey 3. rounds: *a*) SubBytes *b*) ShiftRows *c*) MixColumns *d*) AddRoundKey 4. round final: *a*) SubBytes *b*) ShiftRows *c*) AddRoundKey 5. Return State	1. KeyExpansions 2. round inicial: *a*) InvAddRoundKey 3. rounds: *a*) InvShiftRows *b*) InvSubBytes *c*) InvAddRoundKey *d*) InvMixColumns 4. round final: *a*) InvShiftRows *b*) InvSubBytes *c*) InvAddRoundKey 5. Return State

Los nombres utilizados a continuación son los mismos que los utilizados en la bibliografía y en el pseudocódigo, con el fin de mantener la consistencia.

Variables utilizadas:

- State: arreglo de 16 bytes que puede ser pensado como una matriz de 4×4 bytes. Es el arreglo que contendrá el input y que será modificado hasta contener el valor final cifrado (al ser un algoritmo simétrico que usa bloques de 128 bits, su output también es de 128 bits).
- s-Box: llamadas substitution-box, es una matriz de tamaño 16×16 bytes. Su funcionamiento es simple, en base a un par de coordenadas x e y, se toma el valor correspondiente de la matriz.
- Inverse s-Box: Es una s-Box como la anterior, pero contiene otros valores que funcionarán a la inversa, permitiendo el descifrado del bloque.

- Nk: es el número de double words (32 bits) que conforman la clave original. Luego, solo puede tomar los valores 4, 6 o 8.
- Nr: es el número de rounds con los que operará el algoritmo, pudiendo ser 10, 12 o 14 según el tamaño de la clave (Nr:=Nk+6).
- Nb: es el número de "words" en un bloque (la especificación oficial considera words de 32 bits, sin embargo, en la arquitectura utilizada en el presente trabajo estas son llamadas double words).
- w: es el arreglo donde se guardará la clave expandida. Su tamaño está dado por la siguiente función: $4 \times Nb \times (Nr + 1)$ bytes
- wCount: es un contador auxiliar que será utilizado para saber con que posición de la clave expandida se esta trabajando.

Funciones utilizadas:

- KeyExpansion: toma la clave recibida como input y devuelve una clave expandida de 128 bits $\times$ cantidad de rounds + 1 (varía según el tamaño de la clave original). La finalidad de esta función es lograr que cada round pueda utilizar una clave diferente con el fin de aumentar la seguridad.
- Para cifrado:
  1. AddRoundKey: aplica un XOR entre la matriz state y una porción de la clave expandida (varia segun el numero de round que se esta procesando).
  2. SubBytes: se aplica una sustitución no lineal (cada byte contenido en state es reemplazado) mediante el uso de la s-Box.
  3. ShiftRows: se aplica un shift circular derecho de las filas 1, 2 y 3 de la matriz State. El valor de los desplazamientos es de 1, 2 y 3 bytes correspondientemente.
  4. MixColumns: realiza una operación de mezclado combinando los bytes correspondientes a cada columna de la matriz State mediante el uso de transformaciones lineales inversibles.
- Para descifrado:
  1. InvAddRoundKey: realiza la misma función que AddRoundKey pero en sentido inverso.
  2. InvShiftRows: realiza lo inverso a ShiftRows, es decir, un shift circular izquierdo.
  3. InvSubBytes: realiza las mismas operaciones que SubBytes pero utilizando la matriz s-Box inversa (con lo cual se logra el efecto inverso a SubBytes).
  4. InvMixColumns: aplica la transformación inversa de MixColumns.

Modos de utilización:
Como se explicó anteriormente, según el tamaño de la clave ciertas variables deben ser inicializadas con ciertos valores específicos. La siguiente tabla presenta dichos valores (en words de 32 bits):

Modo de operacion	Tamaño de la clave (Nk)	Cantidad de rounds (Nr)	Tamaño de la clave expandida (Nb x (Nr+1))
AES-128	4	10	44
AES-192	6	12	52
AES-256	8	14	60

4.2.2. Pseudocódigo: cifrado

Algorithm 6 Cifrado

```
1: function CIPHER(input, key)
2:     var State:= input, w:= keyExpansion(key)
3:     AddRoundKey(State, w)
4:     for i from 1 to Nr do
5:         SubBytes(State)
6:         ShiftRows(State)
7:         MixColumns(State)
8:         AddRoundKey(State, w)
9:     end for
10:    SubBytes(State)
11:    ShiftRows(State)
12:    AddRoundKey(State, w)
13:    return State
14: end function
15:
16: function SUBBYTES(State)
17:    for row from 0 to 4 do
18:        for col from 0 to 4 do
19:            State[row][col]:= Sbox[state[row][col]]          ▷ Lecturas a memoria de un byte
20:        end for
21:    end for
22: end function
23:
24: function SHIFTROWS(State)
25:    for row from 1 to 4 do
26:        CircularRightShift(State, row, row×8)
27:    end for
28: end function
29:
30: function ADDROUNDKEY(State)
31:    for row from 0 to 4 do
32:        for col from 0 to 4 do
33:            State[row][col]:= State[row][col] xor w[wCount]  ▷ Lecturas a memoria de un byte
34:            wCount:= wCount + 1
35:        end for
36:    end for
37: end function
38:
39: function MIXCOLUMNS(S)
40:    var SAux[0..3]
41:    for col from 0 to 4 do
42:        SAux[0]:= fMul(0x02, S[0][c]) xor fMul(0x03, S[1][c]) xor S[2][c] xor S[3][c]
43:        SAux[1]:= S[0][c] xor fMul(0x02, S[1][c]) xor fMul(0x03, S[2][c]) xor S[3][c]
44:        SAux[2]:= S[0][c] xor S[1][c] xor fMul(0x02, S[2][c]) xor fMul(0x03, S[3][c])
45:        SAux[3]:= fMul(0x03, S[0][c]) xor S[1][c] xor S[2][c] xor fMul(0x02, S[3][c])
46:        for i from 0 to 4 do
47:            State[i][col]:= SAux[i]
48:        end for
49:    end for
50: end function
```

4.2.3. Pseudocódigo: descifrado

Algorithm 7 Descifrado

```
1: function DECIPHER(input,key)
2:     var State:= input, w:= keyExpansion(key)
3:     InvAddRoundKey(State, w)
4:     for i from 1 to Nr do
5:         InvShiftRows(State)
6:         InvSubBytes(State)
7:         InvAddRoundKey(State, w)
8:         InvMixColumns(State)
9:     end for
10:    InvShiftRows(State)
11:    InvSubBytes(State)
12:    InvAddRoundKey(State, w)
13:    return State
14: end function
15:
16: function INVSUBBYTES(State)
17:    for row from 0 to 4 do
18:        for col from 0 to 4 do
19:            State[row][col]:= InvSbox[state[row][col]]        ▷ Lecturas a memoria de un byte
20:        end for
21:    end for
22: end function
23:
24: function INVSHIFTROWS(State)
25:    for row from 1 to 4 do
26:        CircularLeftShift(State, row, row×8)
27:    end for
28: end function
29:
30: function INVADDROUNDKEY(State)
31:    for row from Nb-1 to 0 do
32:        for col from 3 to 0 do
33:            State[row][col]:= State[row][col] xor w[wCount]    ▷ Lecturas a memoria de un byte
34:            wCount:= wCount - 1
35:        end for
36:    end for
37: end function
38:
39: function INVMIXCOLUMNS(S)
40:    var sA[0..3]
41:    for col from 0 to 4 do
42:        sA[0]:=fM(0x0e, S[0][c]) xor fM(0x0b, S[1][c]) xor fM(0x0d, S[2][c]) xor fM(0x09, S[3][c])
43:        sA[1]:=fM(0x09, S[0][c]) xor fM(0x0e, S[1][c]) xor fM(0x0b, S[2][c]) xor fM(0x0d, S[3][c])
44:        sA[2]:=fM(0x0d, S[0][c]) xor fM(0x09, S[1][c]) xor fM(0x0e, S[2][c]) xor fM(0x0b, S[3][c])
45:        sA[3]:=fM(0x0b, S[0][c]) xor fM(0x0d, S[1][c]) xor fM(0x09, S[2][c]) xor fM(0x0e, S[3][c])
46:        for i from 0 to 4 do
47:            State[i][col]:= sA[i]
48:        end for
49:    end for
50: end function
```

4.2.4. Pseudocódigo: expansión de clave

Algorithm 8 keyExpansion

```
1:  function KEYEXPANSION(key)
2:      var w[0..(4×Nb×(Nr + 1))]
3:      for i from 0 to Nk do
4:          w[i]:= key[i]                          ▷ Lectura a memoria de 32 bits
5:      end for
6:      for i from Nk to Nb×(Nr + 1) do
7:          aux:= w[i-1]
8:          if i ≡ 0 (Nk) then
9:              aux:= SubWord(RotWord(aux)) xor Rcon[i/Nk]
10:         else if Nk > 6 and (i ≡ 4 (Nk)) then
11:             aux:= SubWord(aux)
12:         end if
13:         w[i]:= w[i-Nk] xor aux
14:     end for
15:     return w
16: end function
```

Auxiliares:

- RotWord: función que realiza una rotación circular izquierda en una word de 32 bits. Ej: RotWord([a, b, c, d]):= [b, c, d, a]
- SubWord: aplica una sustitución en base a la s-Box en cada byte de una word de 32 bits.
- Rcon[i]: arreglo que contiene constantes auxiliares.
- fMul/fM: realiza una multiplicación en un campo de Galois. Con el fin de mejorar la performance, utiliza tablas precomputadas con valores basados en logaritmos y exponenciales.

4.2.5. Detalles y decisiones de implementación

- Constantes pre-computadas: Este algoritmo cuenta con muchas constantes que pueden ser pre computadas con el fin de ahorrar tiempo de cómputo en el momento de la inicialización del algoritmo. Estas constantes son: S-Box, Inv S-Box, rCon, eTable y lTable. Por lo tanto, estas constantes se encontraran pre-computadas en la sección de datos del algoritmo.
- Funcion Cipher/Decipher: Estas funciones no pueden ser optimizadas mediante la utilización de SIMD ya que las mismas están compuestas principalmente por llamados a otras funciones.
- Multiplicación en campo de Galois: Se optimizo esta operación mediante la utilización de tablas precomputadas, reduciendo la cantidad de procesamiento con respecto a otras implementaciones. Dado que la cantidad de almacenamiento requerida para tales tablas es ínfima, esta mejora resulta conveniente.
- KeyExpansion: La sección donde se copia la clave original puede ser optimizada realizando la carga a través de registros XMM/YMM e instrucciones SIMD. Sin embargo, el cuerpo principal de la función (donde se expande la clave) no puede ser optimizado mediante instrucciones SIMD ya que, a medida que itera, necesita el resultado de la iteración anterior (por lo cual no pueden calcularse simultáneamente). Adicionalmente, las funciones llamadas dentro del ciclo tampoco pueden ser paralelizadas.
- ShiftRows: Esta función puede ser optimizada mediante la utilización de instrucciones SIMD. Dado que State ocupa 16 bytes, entra exactamente en un registro XMM. Luego, es posible aplicar una operación de shuffle, logrando realizar todas las operaciones mediante una sola instrucción. El único costo adicional al shuffle es el correspondiente a lecturas y escrituras a memoria, siendo estas también optimizadas mediante instrucciones SIMD para registros XMM.
- addRoundKey: Esta función puede ser optimizada mediante la utilización de instrucciones SIMD. Tanto State como la subclave correspondiente ocupan 16 bytes, luego, caben perfectamente en dos registros XMM. Una vez en tales registros, el XOR entre ambos puede ser realizado mediante una única instrucción.
- SubBytes: Esta función no puede ser optimizada mediante la utilización de instrucciones SIMD. Esto se debe a que cada byte es reemplazado por un valor de la S-Box según su propio valor y este valor no necesariamente es contiguo. Luego, si se quisiera reemplazar dos posiciones contiguas de state, podría requerir realizar lecturas en dos posiciones de memoria no contiguas de la S-Box. SubWord tampoco es paralelizable por el mismo motivo.
- Funciones inversas: las funciones inversas a todas las nombradas previamente son equivalentes en términos de posibilidad o imposibilidad de optimización.
- Carga del input y guardado del resultado: al utilizar un bloque de 16 bytes de tamaño, State puede ser cargado y guardado en una única instrucción mediante registros XMM y instrucciones SIMD.

4.2.6. Bibliografía y referencias

- Neal R. Wagner, *The Laws of Cryptography with Java Code*, 2003.
  Disponible en: $http://www.cs.utsa.edu/\ wagner/lawsbookcolor/laws.pdf$.
- Harald Baier, *Advanced Encryption Standard (AES)*, 2010.
  Disponible en: $https://goo.gl/rL7pY3$.
- Wikipedia, *Advanced Encryption Standard*.
  Disponible en: $https://en.wikipedia.org/wiki/Advanced\_Encryption\_Standard$.
- Wikipedia, *Rijndael key schedule*.
  Disponible en: $https://en.wikipedia.org/wiki/Rijndael\_key\_schedule$.
- Sam Trenholme, *Rijndael's key schedule*.
  Disponible en: $http://www.samiam.org/key-schedule.html$.

5. Testing

5.1. Fundamentos

En esta etapa se busca verificar el correcto funcionamiento de todos los algoritmos previamente mostrados. Dada la naturaleza de los mismos, no es computacionalmente viable verificar su funcionamiento para cada posible input, ya que esto sería equivalente a un ataque por fuerza bruta y, por ende, a vulnerar la seguridad de dichos algoritmos. Sin embargo, dadas las siguientes propiedades (comunes a los algoritmos de hashing y cifrado), será posible realizar el testeo sobre una menor cantidad de datos (haciéndolo computacionalmente posible):

1. Son determinísticos, para un input dado, siempre generarán el mismo resultado.
2. Los resultados deben ser muy diferentes para inputs similares. Luego, modificar cualquier bit del input debería cambiar drásticamente el resultado final.
3. Es inviable computacionalmente encontrar dos inputs diferentes que generen el mismo resultado.

A partir de estas propiedades se desprende lo siguiente:

- Por la propiedad 2: dado que el cambio de un bit (en cualquier paso) generaría el cambio del resultado final (por la naturaleza de estos algoritmos), si hubiese algún error en el algoritmo, el resultado debería variar drásticamente con respecto al correcto.
- Por la propiedad 3: dado que la probabilidad de que dos inputs diferentes den el mismo output es extremadamente baja, la probabilidad de obtener un valor correcto si hubiese un error en el algoritmo, también es extremadamente baja.

Las consecuencias de tales propiedades afectan tanto a la etapa de padding (en caso de ser incorrecta le proporcionaría un input incorrecto al algoritmo de cálculo), como a la etapa de cálculo (cualquier bit mal calculado en una etapa intermedia generaría el mismo efecto que recibir otro input, ya que todos los pasos influyen en el resultado final).

Por lo tanto, si se contrasta contra un valor correcto y el resultado obtenido es incorrecto, sabemos que hay un error en el algoritmo. En cambio, si se contrasta con un valor correcto y el resultado obtenido es correcto, si bien no podemos garantizar que no haya un error, sabemos que la probabilidad de que esto ocurra es muy baja. Luego, cuantos más datos se prueben, mayor certeza se tendrá del correcto funcionamiento. Por lo tanto, se contrastarán los resultados provistos por esta librería contra los resultados provistos por otra implementación (ampliamente testeada y validada) para una gran cantidad de datos.

5.2. Programas auxiliares

Para realizar el testing se implementaron los siguientes programas auxiliares:

- Generador de casos de test: implementado en Java, este programa genera archivos de texto con inputs de tamaño y contenido aleatorios de distribución uniforme. El fin de generar casos de test variando la longitud de los inputs, consiste en verificar el correcto funcionamiento de la etapa de padding de los algoritmos en diferentes contextos.

  En el caso de los algoritmos de cifrado, los inputs de prueba son guardados junto con una clave, y el tamaño de dicho input se corresponde con su tamaño de bloque.

- Generador de output (Java): para cada uno de los archivos de texto (correspondientes a cada algoritmo), se aplica el algoritmo correspondiente sobre cada test (linea) y guarda el resultado en otro archivo con el fin de verificarlos posteriormente. Se utilizo el lenguaje Java ya que implementa muchos de los algoritmos en cuestión nativamente. En el caso de los algoritmos de cifrado, se utilizo la librería “Bouncy Castle”. En ambos casos, las funciones utilizadas cuentan con gran aceptación y confiabilidad.

- Generador de output (C): para cada uno de los archivos de texto (correspondientes a cada algoritmo), aplica el algoritmo correspondiente sobre cada test contenido y guarda el resultado en otro archivo con el fin de verificarlos posteriormente. Este programa utiliza la librería implementada en el contexto del presente trabajo.
- Validador de resultados: implementado en Java, se encarga de verificar que todos los resultados generados por ambas implementaciones sean iguales.

5.3. Conclusión

Luego de verificar positivamente los resultados de ambas implementaciones utilizando sets de 100.000 casos de test para cada algoritmo, puede afirmarse con un gran grado de certeza el correcto funcionamiento de los algoritmos implementados en esta librería.