Implementación de Simon (NSA 2013) en C#

Juan Guillermo Callapiña López
Asignatura Criptografía – Docente: Mg. Jorge Eterovic
Universidad Nacional de La Matanza – Departamento de Ingeniería e Investigaciones Tecnológicas, Florencio Varela 1903 (B1754JEC), San Justo, (5411) 4480-8900
guillekallap@gmail.com

Resúmen

La Agencia de Seguridad Nacional de los Estados Unidos (NSA, por sus siglas en inglés) desarrolló la familia SIMON de cifrados en bloque de peso ligero como ayuda para asegurar aplicaciones en entornos muy restringidos donde AES podía ser adecuado. El presente paper está destinado al ámbito puramente académico para demostrar la implementación de los novedosos algoritmos de criptografía ligera, puntualmente el ya mencionado Simon.

Palabras Clave: *Criptoanálisis, Criptografía Ligera Feistel, SIMON, NSA.*

1. Introducción

El cifrado es parte de nuestra vida cotidiana, principalmente para anticipar el monitoreo de nuestras comunicaciones como llamadas telefónicas, Internet, asociaciones de sistemas de seguridad, haciendo posible el comercio electrónico, cuentas bancarias y demás particularidades que tengan espacio en la informática.

De acuerdo con la historia del avance, las antiguas funciones de cifrado han sido quebradas y se han reconstruido nuevas formas de funciones de cifrado.

La NSA comenzó a trabajar con las cifras de Simon y Speck en 2011. La agencia anticipó que algunas agencias en el gobierno federal de los EE. UU. Necesitarían un cifrado que funcionaría bien en una colección diversa de dispositivos de Internet de las cosas, manteniendo un nivel de seguridad aceptable, dando origen a SIMON (NSA 2013).

2. Reseña del cifrado de bloque ligero

La función básica de los cifrados de bloque es proporcionar seguridad en datos almacenados o información del tercero utilizando métodos de cifrado simétricos. Realizan la operación en bloques de texto fijo de tamaño fijo, lo que da como resultado un bloque de texto cifrado para cada uno. Los tamaños de bloque más utilizados en las cifras de bloque actuales son 64 bits y 128 bits. Los cifrados de bloque contienen dos funciones: función de cifrado y función de descifrado. Los bloques de datos de bytes con tamaño fijo se procesan mediante cifrados de bloque. El uso de una clave de cifrado de bloque se usa para los extremos del receptor y del remitente del canal. Con la utilización de la misma clave para cifrar los bloques de texto simple, da como resultado el mismo bloque de texto cifrado.

SIMON es una familia de cifrados de bloque Feistel livianos y balanceados para que el hardware de alto rendimiento resuelva los problemas de seguridad de los dispositivos altamente restringidos.

Como los algoritmos criptográficos existentes fueron diseñados en gran medida para satisfacer las necesidades de la computación de escritorio, no son particularmente adecuados para su uso en las aplicaciones ligeras usados en sistemas de computación generalizada. La familia SIMON (junto con la familia SPECK) ha sido propuesta para abordar la creciente necesidad de criptografía ligera y flexible.

3. Consideraciones Generales

El cifrado de bloque SIMON es un cifrado Feistel equilibrado con una palabra de ‘n’ bits, y por lo tanto la longitud del bloque es 2n. La longitud de la clave es un múltiplo de n por 2, 3 o 4, que es el valor m. Por lo tanto, una implementación de cifrado Simon se denota como Simon2n / nm.

Por ejemplo, Simon64 / 128 se refiere al cifrado que opera en un bloque de texto plano de 64 bits (n = 32) que usa una clave de 128 bits. El componente de bloque del cifrado es uniforme entre las implementaciones de Simon; sin embargo, la lógica de generación de claves depende de la implementación de 2, 3 o 4 claves.

La programación de keys de SIMON emplea una secuencia de rondas de constante 1bit z_i (representada en little-endian) con el fin de eliminar las propiedades de movimientos y simetrías de desplazamiento circular. El bit constante opera en un bloque de keys una vez por ronda sobre el bit más bajo para agregar entropía no dependiente

de la key a la programación de keys. La secuencia lógica de las constantes de bits se establece mediante el valor de los tamaños de clave y bloque.

z0=11111010001001010110000111001101111101000100101011000011100110
z1=10001110111110010011000010110101001110111110011000010110100
z2=10101111011100000011010010011000101000010001111110010110110011
z3=11011011101011000110010111100000010010001010011100110100001111
z4=11010001111001101011011000100000010111000011001010010011101111

Figura 1. Valores de constante z_i

SIMON admite las siguientes combinaciones de tamaños de bloque, tamaños de clave y número de rondas:

Tamaño de bloque(bits) 2n	Tamaño de Key (bits) nm	Tamaño de Palabra (bits) n	Keys m	Secuencia de constante *zi*	Rondas T
32	64	16	4	z0	32
48	72	24	3	z0	36
	96		4	z1	36
64	96	32	3	z2	42
	128		4	z3	54
128	128	64	2	z2	68
	192		3	z3	69
	256		4	z4	72

Tabla 1. Parámetros de SIMON

4. Algoritmo

En este paper elegí SIMON con 128 de tamaño de bloque como configuración de cifrado para las implementaciones de arquitectura. Con este tamaño de bloque es fácilmente extensibles a SIMON 128/128, SIMON 128/192 o SIMON 128/256, además ofrecen un nivel de seguridad totalmente adecuado ya que la cantidad de rondas van a ser mayores y le costará al atacante romper la clave (Cantidad de Rondas para atacar 53/72 (74%) Complejidad del tiempo 2^{248}) y la NSA ha aprobado Simon128/256 para su uso en los Sistemas de seguridad nacional de EE. UU. En el resto del paper, los parámetros SIMON serán los siguientes:

- Tamaño del bloque: 128
- Tamaño de clave: 128
- Tamaño de palabra: 64
- Palabras clave: 4
- Secuencia constante: z4
- Rondas: 72

Típicamente SIMON utiliza la conocida regla de actividad Feistel. El algoritmo está diseñado para ser extremadamente pequeño en hardware y fácil de serializar en varios niveles, pero se tuvo cuidado para no sacrificar el rendimiento del software. La función principal posee dos partes. En primer lugar, utiliza una función de expansión que crea m bloques de tamaño nm bytes partiendo de la clave o key, para luego utilizarlo en la función de rondas SIMON.

La función de rondas SIMON es una construcción AND-RX con una estructura Feistel equilibrada que utiliza las siguientes operaciones:

- XOR a nivel de bits, denotado como $x \oplus y$.
- AND a nivel de bits, denotado como $x \& y$.
- Rotación de bit a la izquierda ROL, denotado como $S^y(x)$ donde y es el recuento de rotación.

La función de rondas de SIMON (utilizada para el cifrado) se puede expresar como:

$$R_k(x, y) = (y \oplus f(x) \oplus k, x)$$

Donde $f(x) = (S^1x \,\&\, S^8x) \oplus S^2x$
Donde k es la ronda de la key.

[1]La función ronda inversa, utilizada para el descifrado, es:

$$R^{-1}{}_k(x, y) = (y, x \oplus f(y) \oplus k)$$

Para la programación de las keys de SIMON se toman una key y de ella generan una secuencia de keys T palabras $k_0, ..., k_{T-1}$, donde T es el número de rondas. La composición es $R_{kT-1}, \ldots, R_{k1}, R_{k0}$, leído de derecha a izquierda. La Figura 2 muestra el efecto de la función redonda R_{ki} en las dos palabras de subcifrador (x_{i+1}, x_i), siendo x e y mencionados anteriormente, en el i^{th} paso de este proceso.

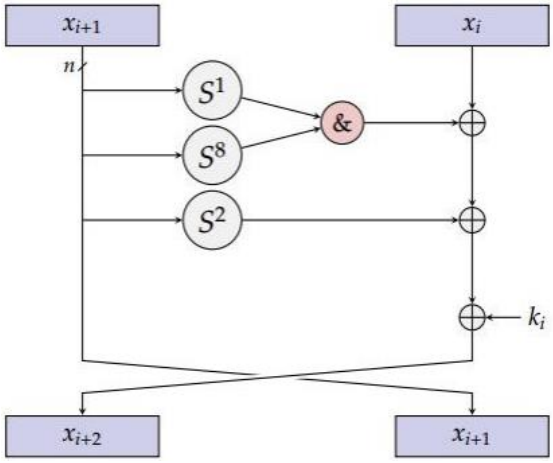

Fig 2. Función de Rondas Simon

Se puede ver que SIMON no incluye pasos de blanqueo de texto simple y de texto cifrado, ya que la inclusión de tales operaciones puede afectar negativamente al tamaño del circuito. En consecuencia, la primera y la última ronda no hacen nada criptográficamente, aparte de actuar para introducir la primera y la última vuelta de keys.

Hay varios otros lugares en la Figura 2 donde se podría haber incluido una key en su lugar, algunas de las cuales no tienen este problema, pero ninguna sin inconvenientes.

Expansión de las keys

Se debe tener en cuenta que aparte de la ronda de key, todas las rondas de SIMON son exactamente iguales, y las operaciones son perfectamente simétricas con respecto al desplazamiento circular en palabras de 'n' bits. Las programaciones de las keys en SIMON emplean una secuencia de 1 bit específicamente para eliminar propiedades de movimientos y simetrías de desplazamiento circular. De hecho, hay ciertas separaciones criptográficas entre diferentes versiones de SIMON que tienen el mismo tamaño de bloque al definir cinco de estas secuencias: z0, z1, z2, z3, z4.

En nuestra configuración utilizamos (representado en little-endian):

z4=11011011101011000110010111100000010010001010011100110100001111

Además, la programación de la key emplea la constante $c = 2^n - 4 = 0xFF...$ terminando en FC donde 'n' es el parámetro de tamaño de palabra, por lo tanto, $c = 2^{64} - 4$ en ésta configuración.

La programación de keys de SIMON proporciona capacidades de expansión de las keys al generar posteriormente todas las rondas de la key maestra. En nuestra configuración SIMON128/256 elegida, la programación de keys genera 72 rondas de keys de tamaño de 64 bits a partir de la clave maestra inicial de 256 bit, este bloque debe ser de mayor o igual tamaño que el bloque que se va a cifrar.

Se realiza, para una ronda dada, combinando las n vueltas de keys anteriores almacenadas en caché (donde n es el parámetro de palabra de keys) con una constante c y una constante de ronda de 1 bit. La función de expansión de key utiliza las siguientes operaciones:

- XOR a nivel de bits, denotado como $x \oplus y$.
- Rotación derecha a modo de bit ROR, denotada como $S^{-y}(x)$ donde y es el recuento de rotación.

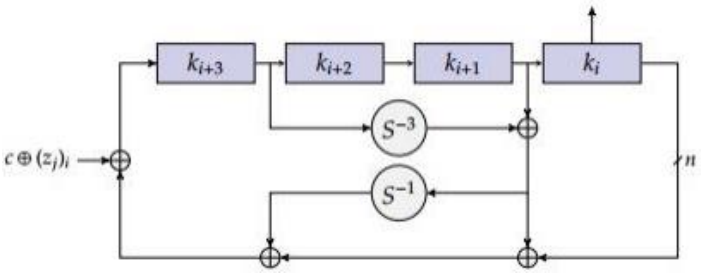

Fig 3. Expansión de keys SIMON de 4 palabras

$k_{i+m} =$

$c \oplus (z_j)_i \oplus k_i \oplus (I \oplus S^{-1}) S^{-3} k_{i+1}$ si m = 2

$c \oplus (z_j)_i \oplus k_i \oplus (I \oplus S^{-1}) S^{-3} k_{i+2}$ si m = 3

$c \oplus (z_j)_i \oplus k_i \oplus (I \oplus S^{-1}) (S^{-3} k_{i+3} \oplus k_{i+1})$ si m=4

para $0 \leq i < T - m$.

Las programaciones de las keys se representan en la Figura 3 o 4, y la elección de la secuencia constante z_j,va a depender de los parámetros en la Tabla 1.

Se debe tener en cuenta que las palabras de la key k_0 a k_{m-1} se utilizan como las primeras m rondas de key; se cargan en los registros de desplazamiento con k_0 a la derecha y k_{m-1} a la izquierda.

Recordemos que 'm' es la cantidad de key, 'j' es la secuencia de la constante z, 'i' es el número de iteracion o constante de ronda.

Al realizar con otras cantidades de key, la Fig. 3. cambiaría a lo que se representa en la Fig. 4.

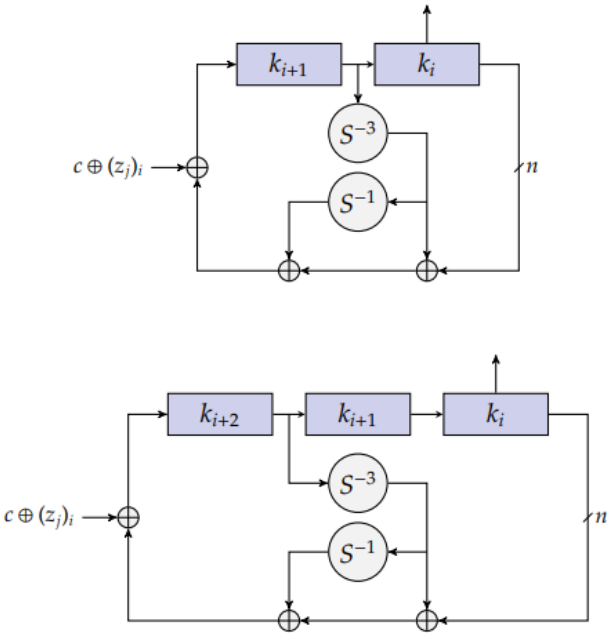

Fig 3. Expansión de keys SIMON de 2 palabras (en la primera) y 3 palabras (en la segunda)

Encriptación

El cifrado de un bloque de texto sin formato de 128 bits consiste simplemente en 72 funciones de vueltas con la respectivas keys producidas por el programa de keys. Debido a la naturaleza de las funciones de vueltas y expansión de keys pueden ejecutarse en paralelo si es así deseado.

Desencriptación

El descifrado de un bloque de texto cifrado de 128 bits *c* consiste en primero intercambiar las palabras de 64 bits de la izquierda y la derecha, seguidas por 72 funciones de vueltas, pero con las vueltas de keys en orden inverso (es decir, rondas de key que comiencen y continúen en 43, ..., 0) seguidas de un cambio final de las palabras de la izquierda y las palabras de más a la derecha.

5. Implementación

A la hora de implementar el algoritmo en un software decidí realizarlo en C# para probar la efectividad de este algoritmo criptográfico [4]. Comencé primero buscando los tipos de dato dentro del lenguaje de programación para poder representar así tanto las palabras de 64 bits como los bloques de mensaje y de expansión.

En C# se dispone de un tipo de datos de 64 bytes sin signo llamado ulong, tuve que utilizarlo para representar los bloques de mensaje, de expansión y las keys utilice un array de ulong, que es un tipo primitivo de C#.

La implementación comprende de una parte gráfica, que permite al usuario ingresar la key, seleccionar el tamaño de la key junto al tamaño del bloque, y encriptar tanto texto plano como archivos; y de la clase Simon, que contiene toda la lógica del cifrado con el algoritmo SIMON 128/ 128-192-256 según lo que haya elegido el usuario, ya sea el tamaño del bloque, de key, cantidad de keys, constante secuencia y rondas. Sobre la clase mencionada se detallará el desarrollo a lo largo de este paper.

Para desarrollar el software cifrador, primero se han separado las funciones del propio algoritmo. A continuación, detallaré cómo se implementó cada función de Simon en C#.

Función de expansión (getExpansionBlock)

Esta función tiene como entrada un array de ulong. Se decidió que el tamaño de la key pueda ser decidido por el usuario ya que la función de expansión no requiere que la key tenga exactamente 16, 24, 32, 64 bytes, sino que la función rellena la clave con ceros hasta lograr el tamaño indicado. En caso de que no se haya indicado un tamaño de key, la función agrega la cantidad de ceros mínima para obtener alguno de estos tamaños, y así realizar el resto de las operaciones.

La función de expansión recibe la clave como un array de ulong, que es un tipo de dato entero de 64 bytes sin signo, y utiliza los datos de la clase ya mencionada devolviendo un array de keys generadas.

Para esto, se utiliza la función **keyExpansion**, que al tener listas las entradas de la función de expansión, instanciamos la salida como un array de ulong de tamaño igual a la cantidad de rondas (t) y almacenamos en ella, el bloque de keys que se dividirá según la cantidad de keys que deseaba el usuario.

Creamos un ulong para la constante de secuencia correspondiente a la selección del usuario (z4) y una variable auxiliar que llamaremos 'tmp'.

En este caso, como vamos a trabajar con bloques de mensajes de tamaño 128bits y bloques de key de tamaño 256bits, vamos a realizar la Expansión de keys SIMON de 4 palabras.

```
//Funcion de expansion de key
private ulong[] keyExpansion(ulong[] key)
{
…
 ulong[] keys = new ulong[t];
 for(int i = 0; i<m; i++)
 {
    keys[i] = key[i];
 }
  ulong zet = z[j];
  ulong tmp;
  for(int i = m; i<t; i++)
  {
      tmp = rotl(key[i - 1], 3);
      if (m == 4) tmp = tmp ^ key[i-3];
      tmp = tmp ^ rotl(tmp, 1);
      keys[i]= key[i - m] ^ tmp ^
        getNBits(rotl(zet,i-m)) ^
        (ulong.MaxValue -3);
   }
    return keys;
  }
```

Podemos apreciar que las funciones requeridas para S^y, es decir los shift rotate, lo realiza en la función **rotl** donde recibe un tipo de dato ulong para el bloque y un numero entero para indicar las posiciones a moverse desde la izquierda. En cuanto a los XOR, utilizo la función **'^'**que recibe como entrada 2 ulong, uno de izquierda y otro de derecha. Para recorrer la expansión de keys partimos por la cantidad de keys (4) hasta llegar a las rondas correspondientes (en nuestro caso serán 72).

A continuación, guardamos en tmp el valor al realizar shift rotate de 3 bits sobre el valor de la key en posición actual-1. Luego verificamos si nuestra cantidad de keys son iguales a 4, por lo que resulta que sí, entonces guardamos en tmp su valor aplicando XOR al valor de la key en posición actual-3. Por consiguiente, tmp se verá reemplazado por su valor aplicando un XOR sobre el valor del shift rotate del mismo tmp en 1 bit. Se resuelve la última operación de la expansión de keys que se trata de guardar en el valor de la key a enviar, la key actual restando la cantidad de keys aplicada a un XOR sobre el valor de tmp aplicado al XOR sobre la función **getNBits()** que devuelve un ulong (luego hablaremos de ella) y aplicado también un XOR a 'c' (0xFF... terminando en FC). Por último, se devuelve el listado de keys para resolver la encriptación del bloque.

Antes de empezar a explicar la encriptación del bloque, debo hacer un paréntesis sobre la función getNBits que recibe como entrada un ulong y devuelve el ulong aplicando a la entrada un shift rotate de 62 – el tamaño del bloque / 2.

Función de encriptación de bloque (cifrarBloque)

Es momento de utilizar las funciones definidas hasta ahora para realizar el cifrado de un bloque. Para esto va a recibir como entrada un array de byte y devuelve también un array de byte.

```
public byte[] cifrarBloque(byte[] bloque)
{
   ulong[] msg = byteToUlongs2(bloque);
   for (int i = 0; i < t; i++)
   {
     ulong tmp = msg[1];
     msg[1]=msg[0]^(rotl(msg[1],1)+rotl(msg[1],8))
          ^ rotl(msg[1], 2) ^ keysGlobal[i];
      msg[0] = tmp;
   }
   return ulongToByte(msg);
}
```

Podemos apreciar que en un array de ulong llamado "msg" guardamos la conversión a ulong del bloque de array de byte que recibimos como entrada, para trabajar con el bloque de datos con el tipo de dato correcto.

Voy a recorrer iteraciones iguales a la cantidad total de rondas que van a utilizarse para la cantidad de keys según el tamaño de key que definió el usuario, donde realizaremos como una función sustitución en 3 instrucciones para obtener los datos del bloque cifrados. Primero, creo una variable ulong 'tmp' y le asigno el segundo valor del msg que hemos convertido (llámese msg2).

Segundo, a ese valor le asignamos el valor del primer bloque del mensaje (llámese msg1) aplicándole un XOR con un shift rotate de 1bit al msg2, al resultado de ambos le aplico un AND entre el shift rotate de 8bit al msg2 aplicado a un XOR al shift rotate de 2bit al msg2 y aplicado a otro XOR sobre la variable keysGlobal[i] que utiliza la key actual de la ronda.

Por último, guardamos en el valor de msg1 el valor de tmp que sería el primer valor del msg2. Cumpliendo con la función de rondas SIMON (véase Fig.2), solo resta enviar el array de ulong que contiene los bloques cifrados pero convertido a array de byte.

Función de desencriptación de bloque (descifrarBloque)

A la hora de utilizar la desencriptación del mensaje, va a recibir de entrada, al igual que la encriptación, un array de byte para devolver otro con los bloques cifrados.

```
public byte[] descifrarBloque(byte[] bloque)
{
    ulong[] msg = byteToUlongs2(bloque);
    for (int i = t - 1; i >= 0; i--)
    {
       ulong tmp = msg[0];
       msg[0] = msg[1] ^ keysGlobal[i] ^ rotl(msg[0],2)
            ^ (rotl(msg[0], 1) + rotl(msg[0], 8));
        msg[1] = tmp;
    }
    return ulongToByte(msg);
}
```

Almacenamos la conversión a ulong del bloque de array de byte que recibimos como entrada, para trabajar con el bloque de datos con el tipo de dato correcto (msg).

Recorremos iteraciones iguales a la cantidad total de rondas que van a utilizarse para la cantidad de keys según el tamaño de key definido, en este caso deberìa ser el mismo que se utilizó en el cifrado, donde realizaremos como una función sustitución en 3 instrucciones para obtener los datos del bloque cifrados. Primero, creo una variable ulong 'tmp' y le asigno el primer valor del msg que hemos convertido (llámese msg1).

Segundo, a ese valor le asignamos el valor del segundo bloque del mensaje (llámese msg2) aplicándole un XOR sobre la variable keysGlobal[i] que utiliza la key actual de la ronda, también aplicada con un XOR al shift rotate de 2bit al msg1 y otro XOR al shift rotate de 1bit al msg1, entre el resultado dado y el shift rotate de 8bit al msg2 aplico un AND.

Por último, guardamos en el valor de msg2 el valor de tmp que sería el primer valor del msg1. Cumpliendo con la [1]función de rondas SIMON inversa, solo resta enviar el array de ulong que contiene los bloques cifrados pero convertido a array de byte.

Encriptar un mensaje completo

Para encriptar cualquier mensaje simplemente se debe separar el mismo en bloques de 64 bits y realizar la función cifrarBloque(), utilizando las keys e incrementando el número de bloque por cada bloque cifrado.
Dispongo en una clase Simon una función para encriptar un array de ulong y otra para encriptar un archivo que indican tanto la ruta del **archivo** a cifrar como la del archivo cifrado, acompañado de funciones de conversión para tratar ambos casos al cambiar el tipo de dato.

6. Conclusiones

Como respuesta para la finalización de este paper, mencionaré algunas conclusiones de mi experiencia implementando SIMON en C#.

SIMON

Es un algoritmo interesante cuando se tiene que implementar y ofrece seguridad ya que posee bastantes vueltas a la hora de querer atacar archivos encriptados con este algoritmo. El criptoanálisis de este algoritmo escapa de lo que trasmití en este paper, pero mi idea fue enfocarme en cómo utilizarlo de manera didáctica, pude reconocer que está diseñado para tener la máxima seguridad posible para cada bloque y tamaño de key, contra ataques estándar de texto sin formato (CPA) y texto cifrado (CCA). Considero que se debe utilizar bloques grandes de datos, como use en este trabajo, para así evitar que se descifre fácilmente el algoritmo y la dificultad aumente obligando el uso de otras herramientas de descifrado, realice con bloques grandes para que pueda perdurar al menos para algunos proyectos que tenga mente.

Proposiciones o mejoras para el futuro

La implementación que realice puede verse ampliamente mejorada debido a la falta de tiempo, he aquí algunos detalles a tener en cuenta:

• SIMON se puede implementar en diversas arquitecturas de hardware y opciones de diseño asociadas, soluciones a problemas y compensaciones. Puede ser implementada por ejemplo para el uso de la matriz de puerta programable de campo (FPGA) ya que consisten en una multitud de segmentos universales reconfigurables que se conectan en formas reconfigurables.

• SIMON posee su uso en el universo del IoT, per se podrían hacer mejoras a fin de reducir el tiempo de cifrado y mantener el equilibrio práctico entre la seguridad y el rendimiento.

Consideraciones al implementar SIMON

Al momento de implementar este algoritmo en cualquier lenguaje de programación, consideramos que se tengan en cuenta los siguientes puntos:

- Se debe evaluar si se dispone de las tres operaciones base de este algoritmo, sino simular con los operandos que proporcione el lenguaje de programación.
- Admite cifrado paralelo de bloques para así reducir tiempos de cifrado. Por la propia naturaleza del algoritmo, el cifrado de un bloque no depende de ningún otro.
- Se debe pensar sobre la forma en representar las palabras por las cuales se divide tanto el bloque de expansión como el bloque de mensajes. Recomiendo utilizar un tipo de dato entero de 64 bits sin signo, ya que otro tipo de dato dificultaría la programación y rotación de bit. En caso de que el lenguaje no disponga de un tipo de dato similar, se deberá evaluar como representarlos.

7. Referencias

[1] https://en.wikipedia.org/wiki/Simon_(cipher).

[2] Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L., "Simon and Speck: Block Ciphers for the Internet of Things", en National Security Agency 9800 Savage Road, Fort Meade, MD, 20755, USA, 9 July 2015.

[3] Wetzels, S., Bokslag, W. "*Simple SIMON FPGA implementations of the **SIMON64/128** Block Cipher",* Kent State University, Kent, OH, USA, Doctoral Thesis, 2003.

[4] Repositorio de implementación de software de SIMON NSA 2013 https://github.com/Guillekallap/Simon-NSA2013