GENERACIÓN DE TOKENS PARA PROTEGER LOS DATOS DE TARJETAS BANCARIAS

Trabajo terminal No. 2017-B008

Directora

Dra. Sandra Díaz Santiago

Presentan

Daniel Ayala Zamorano

Laura Natalia Borbolla Palacios

Ricardo Quezada Figueroa

Instituto Politécnico Nacional

Escuela Superior de Cómputo

Contenido

Simbología

A continuación se describe la simbología que se utilizará a lo largo de este documento.

Tabla 1: Simbología

Símbolo	Descripción
K	Llave
pk	Llave pública
sk	Llave privada o secreta
k_i	I*é*-sima subllave
K_I	Llave de entrada o *key derivation key* de una función de derivación de llaves
K_O	Material de llaves obtenido de una función de derivación de llaves
IV	Vector de inicialización
E	Operación de cifrado
E_K	Operación de cifrado utilizando la llave K
D	Operación de descifrado
D_K	Operación de descifrado utilizando la llave K
h	Función hash
h_k	Función hash que utiliza una llave k para calcular el valor
M	Mensaje en claro
C	Mensaje cifrado
$\mathbb{Z}_n$	Conjunto de los números enteros módulo n
$\{0,1\}^r$	Cadena de bits de longitud r
$\{0,1\}^*$	Cadena de bits de longitud arbitraria
mód	Operación módulo
mcd	Máximo común divisor
$\oplus$	Operación XOR
$\boxplus$	Suma modular
$\boxminus$	Resta modular
$\|\|$	Operación de concatenación
$\{X\}$	Indicación de que el uso de X es opcional
$[X]$	El entero más pequeño que es mayor o igual que X
$[X]_2$	Representación binaria del número X
φ	Función ϕ de Euler
$\emptyset$	Conjunto vacío

Es menester aclarar que un mensaje no consiste solo en letras y números; el *mensaje* se refiere al conjunto de datos que van a ser cifrados o descifrados.

Capítulo 1

Introducción

1.1. Justificación

1.2. Objetivos

Capítulo 2

Antecedentes

2.1. Introducción a la criptografía

La información presentada a continuación puede consultarse con más profundidad en las siguientes referencias [1], [2].

La palabra criptografía proviene de las etimologías griegas *Kriptos* (ocultar) y *Graphos* (escritura), y es definida por la Real Academia Española (RAE) como «el arte de escribir con clave secreta o de un modo enigmático». De manera más formal se puede definir a la criptografía como la ciencia encargada de estudiar y diseñar por medio de técnicas matemáticas, métodos y modelos capaces de resolver problemas en la seguridad de la información, como la confidencialidad de esta, su integridad y la autenticación de su origen.

2.1.1. Objetivos de la criptografía

La criptografía tiene como finalidad proveer los siguientes cuatro servicios:

Confidencialidad Es el servicio encargado de mantener legible la información solo a aquellos que estén autorizados a visualizarla.

Integridad Este servicio se encarga de evitar la alteración de la información de forma no autorizada, esto incluye la inserción, sustitución y eliminación de los datos.

Autenticación Este servicio se refiere a la identificación tanto de las personas que establecen una comunicación, garantizando que cada una es quien dice ser; como del origen de la información que se maneja, garantizando la veracidad de la hora y fecha de origen, el contenido, tiempos de envío, entre otros.

No repudio Es el servicio que evita que el autor de la información o de alguna acción determinada, pueda negar su validez, ayudando así a prevenir situaciones de disputa.

2.1.2. Criptoanálisis y ataques

La criptografía forma parte de una ciencia más general llamada criptología, la cual tiene otras ramas de estudio, como es el criptoanálisis que es la ciencia encargada de estudiar los posibles ataques a sistemas criptográficos, que son capaces de contrariar sus servicios ofrecidos. Entre los principales objetivos del criptoanálisis están interceptar la información que circula en un canal de comunicación, alterar dicha información y suplantar la identidad, rompiendo con los servicios de confidencialidad, integridad y autenticación respectivamente.

Los ataques que se realizan a sistemas criptográficos dependen de la cantidad de recursos o conocimientos con los que cuenta el adversario que realiza dicho ataque, dando como resultado la siguiente

clasificación.

Ataque con sólo texto cifrado En este ataque el adversario solamente es capaz de obtener la información cifrada, y tratará de conocer su contenido en claro a partir de ella. Esta forma de atacar es la más básica, y todos los métodos criptográficos deben poder soportarla.

Ataque con texto en claro conocido Esta clase de ataques ocurren cuando el adversario puede obtener pares de información cifrada y su correspondiente información en claro, y por medio de su estudio, trata de descifrar otra información cifrada para la cual no conoce su contenido.

Ataque con texto en claro elegido Este ataque es muy parecido al anterior, con la diferencia de que en este el adversario es capaz de obtener los pares de información cifrada y en claro con el contenido que desee.

Ataque con texto en claro conocido adaptativo En este ataque el adversario es capaz de obtener los pares de información cifrada y en claro con el contenido que desee, además tiene amplio acceso o puede usar de forma repetitiva el mecanismo de cifrado.

Ataque con texto en claro elegido adaptativo En este caso el adversario puede elegir información cifrada y conocer su contenido, dado que tiene acceso a los mecanismos de descifrado.

2.1.3. Clasificación de la criptografía

La criptografía puede clasificarse de forma histórica en dos categorías, la criptografía clásica y la criptografía moderna. La criptografía clásica es aquella que se utilizó desde la antigüedad, teniéndose registro de su uso desde hace más 4000 años por los egipcios, hasta la mitad del siglo XX. En esta los métodos utilizados para cifrar eran variados, pero en su mayoría usaban la transposición y la sustitución, además de que la mayoría se mantenían en secreto. Mientras que la criptografía moderna es la que se inició después la publicación de la *Teoría de la información* por Claude Elwood Shannon[3], dado que esta sentó las bases matemáticas para la CRIPTOLOGÍA en general.

Una manera de clasificar es de acuerdo a las técnicas y métodos empleados para cifrar la información, esta clasificación se puede observar en la figura 2.1.

Adentrándose en la clasificación de la criptografía clásica, se tienen los cifrados por transposición, los cuales se basan en técnicas de PERMUTACIÓN de forma que los caracteres de la información en claro se reordenen mediante algoritmos específicos, y los cifrados por sustitución, que utilizan técnicas de modificación de los caracteres por otros correspondientes a un alfabeto específico para el cifrado.

En cuanto a la criptografía moderna, esta tiene dos vertientes: la criptografía simétrica o de llave secreta, y la asimétrica o de llave pública. Hablando de la primer vertiente, se puede decir que es aquella

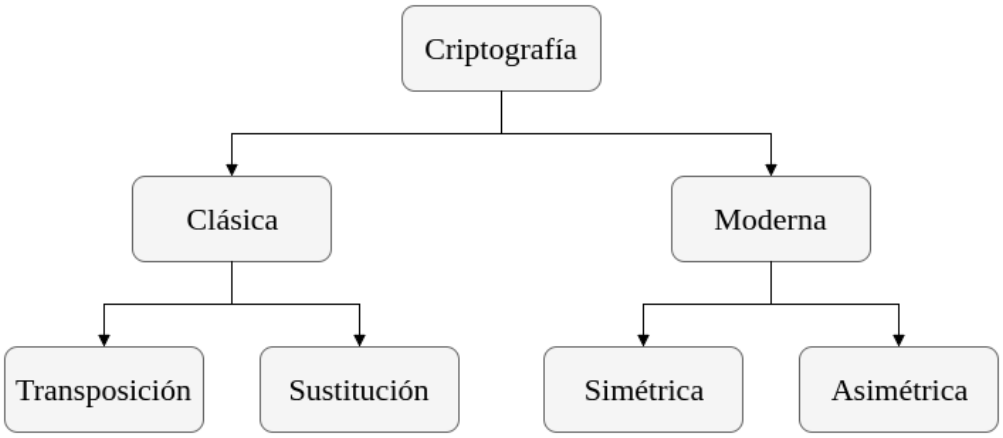


Figura 2.1: Clasificación de la criptografía.

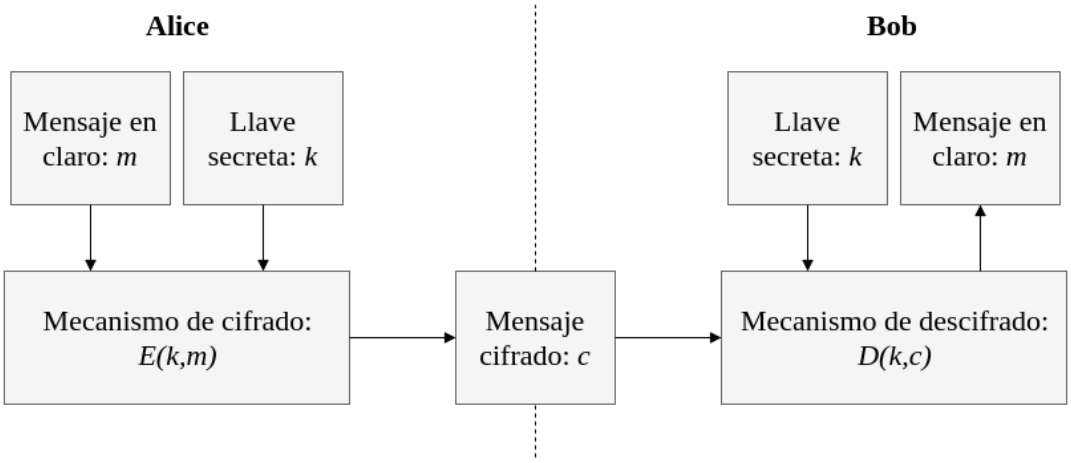


Figura 2.2: Canal de comunicación con criptografía simétrica.

que utiliza un modelo matemático para cifrar y descifrar un mensaje utilizando únicamente una llave que permanece secreta.

En la figura 2.2 se puede observar el proceso para establecer una comunicación segura por medio de la criptografía simétrica. Primero, tanto Alice como Bob deben de establecer una llave única y compartida k, para que después, Alice, actuando como el emisor, cifre un mensaje m usando la llave k por medio del algoritmo de cifrado $E(k, m)$ para obtener el mensaje cifrado c y enviárselo a Bob. Posteriormente Bob, como receptor, se encarga de descifrar c con ayuda de la llave k por medio del algoritmo de descifrado $D(k, c)$ para obtener el mensaje original m.

Gran parte de los algoritmos de cifrado que caen en este tipo de criptografía están basados en las redes Feistel, que son un método de cifrado propuesto por el criptógrafo Horst Feistel, mismo que desarrolló el Data Encryption Standard (DES) (sección 2.2.4) a principios de la década de los 70, que fue el cifrado

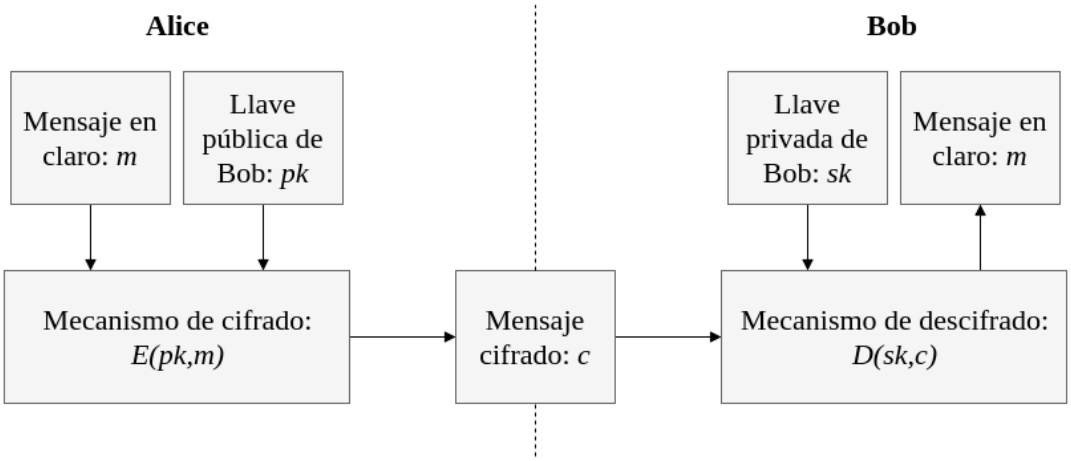


Figura 2.3: Canal de comunicación con criptografía asimétrica.

usado por el gobierno estadounidense hasta 2002, año que el Advanced Encryption Standard (AES) (sección 2.2.5) lo sustituyó.

Ahora, adentrándose en la criptografía asimétrica, se tiene que su idea principal es el uso de 2 llaves distintas para cada persona, una llave pública para cifrar, que esté disponible para cualquier otra persona, y una llave privada para descifrar, que se mantiene disponible solo para su propietario.

El proceso para establecer una comunicación segura por medio de este tipo de criptografía es el siguiente: primero, Alice nuevamente como el emisor, cifra un mensaje m con la llave pública de Bob pk, y usa el algoritmo de cifrado $E(pk, m)$ para obtener c y enviarlo. Después Bob como receptor, se encarga de descifrar c por medio del algoritmo de descifrado $D(sk, c)$ haciendo uso de su llave privada sk. Este proceso se refleja gráficamente el la figura 2.3.

Entre los uso que se le da a esta criptografía está el mantener la distribución de llaves privadas segura y establecer métodos que garanticen la autenticación y el no repudio; por ejemplo, en las firmas y certificados digitales.

El principal precursor de la criptografía asimétrica fue el método de intercambio de llaves de Diffie-Hellman, desarrollado y publicado por Whitfield Diffie y Martin Hellman, en 1976 en el artículo *New Directions in Cryptography*[4], siendo la primera forma práctica para poder establecer una llave secreta compartida entre dos partes sin contacto previo por medio de un canal público para intercambiar mensajes.

Otro precursor fue el sistema criptográfico Ron Rivest, Adi Shamir, Leonard Adleman (RSA) (nombre obtenido por las siglas de los apellidos de sus desarrolladores: Ron Rivest, Adi Shamir y Leonard Adleman), publicado en 1978[5] y que fue el primer sistema criptográfico capaz de servir tanto para cifrar mensajes, como para la implementación de firmas digitales. A pesar de que sus orígenes son de ya hace casi cuatro décadas, este sistema aún es uno de los más ampliamente usados.

```
1  entrada: ninguna.
2  salida: llave pública (n,e) y privada (n,d).
3  inicio
4    Elegir de forma aleatoria 2 números primos p y q, que sean de gran
5    magnitud y de una longitud parecida.
6    Calcular n = p · q
7    Calcular φ(n) = (p−1)(q−1).
8    Elegir un exponente de cifrado e tal que e < φ(n) y mcd(e,φ(n)) = 1.
9    Encontrar el exponente de descifrado d tal que e·d mód φ(n) = 1.
10 fin
```

Pseudocódigo 2.1: Proceso de generación de llaves de RSA.

Entre los motivos del éxito de RSA está que su funcionamiento se basa en la teoría elemental de números, ya que usa propiedades descritas en esta teoría; y en su seguridad, ya que se basa en la incapacidad de poder factorizar números grandes de forma eficiente.

El algoritmo de RSA consta de 3 partes, la generación de llaves, el cifrado y el descifrado. El proceso para poder generar un par de llaves (pública y privada) con RSA se muestra en el pseudocódigo 2.1.

Las funciones de cifrado y descifrado, están definidas en las ecuaciones 2.1 y 2.2 respectivamente, y se aplican números enteros o bloques de bits, siendo funciones biyectivas e inversas entre sí.

$$E : \mathbb{Z}_n \longrightarrow \mathbb{Z}_n, x \longmapsto x^e \tag{2.1}$$

$$D : \mathbb{Z}_n \longrightarrow \mathbb{Z}_n, x \longmapsto x^d \tag{2.2}$$

2.2. Cifrados por bloques

La información presentada a continuación puede consultarse con más profundidad en las siguientes referencias [1], [2].

Los cifrados por bloque son esquemas de cifrado que, como bien lo explica su nombre, operan mediante bloques de datos. Normalmente los bloques tienen una longitud de 64 o de 128 bits, mientras que las llaves pueden ser de 56, 128, 192 o 256 bits.

En muchos sistemas criptográficos, los cifrados por bloque simétricos son elementos importantes, pues su versatilidad permite construir con ellos generadores de números pseudoaleatorios, cifrados de flujo MACs y funciones hash. Sirven también como componentes centrales en técnicas de autenticación de mensajes, mecanismos de integridad de datos, protocolos de autenticación de entidad y esquemas de firma electrónica que usan llaves simétricas.

Los cifrados por bloque están limitados en la práctica por varios factores, tales como el límite de memoria, la velocidad requerida o restricciones impuestas por el hardware o el software en el que se implementan. Normalmente, se debe escoger entre eficiencia y seguridad

Idealmente, al cifrar por bloques, cada bit del bloque cifrado depende de todos los bits de la llave y del texto en claro; no debería existir una relación estadística evidente entre el texto en claro y el texto cifrado; el alterar tan solo un bit en el texto en claro o en la llave debería alterar cada uno de los bits del texto cifrado con una probabilidad de $\frac{1}{2}$; y alterar un bit del texto cifrado debería provocar resultados impredecibles al recuperar el texto en claro.

2.2.1. Definición

$$\begin{aligned} E : \{0,1\}^r \times \{0,1\}^n &\longrightarrow \{0,1\}^n \\ (k, m) &\longmapsto E(k, m) \end{aligned} \tag{2.3}$$

Utilizando una llave secreta k de longitud binaria r el algoritmo de cifrado E cifra bloques en claro m de una longitud binaria fija n y da como resultado bloques cifrados $c = E(k, m)$ cuya longitud también es n. n es el tamaño de bloque del cifrado. El espacio de llave está dado por $K = \{0,1\}^r$, para cada llave existe una función $D_k(c)$ que permite tomar un bloque cifrado c y regresarlo a su forma original m.

Generalmente, los cifrados por bloque procesan el texto claro en bloques relativamente grandes ($n \geq 64$), contrastando con los cifradores de flujo, que toman bit por bit. Cuando la longitud del mensaje en claro excede el tamaño de bloque, se utilizan los MODOS DE OPERACIÓN.

Los parámetros más importantes de los cifrados por bloque son los siguientes:

```
1  inicio
2  para_todo i desde 1 hasta r:
3     L_i = R_{i-1}
4     R_i = L_{i-1} ⊕ f(R_{i-1}, K_i)
5  fin
6  fin
```

Pseudocódigo 2.2: Feistel, cifrado.

- Tamaño de bloque
- Tamaño de llave

2.2.2. Criterios para evaluar los cifrados por bloque

A continuación se listan algunos de los criterios que pueden ser tomados en cuenta para evaluar estos cifrados:

1. **Nivel de seguridad.** La confianza que se le tiene a un cifrado va creciendo con el tiempo, pues va siendo analizado y sometido a pruebas.
2. **Tamaño de llave.** La ENTROPÍA del espacio de la llave define un límite superior en la seguridad del cifrado al tomar en cuenta la búsqueda exhaustiva. Sin embargo, hay que tener cuidado con su tamaño, pues también aumentan los costos de generación, transmisión, almacenamiento, etcétera.
3. **Tamaño de bloque.** Impacta la seguridad, pues entre más grandes, mejor; sin embargo, tiene repercusiones en el costo de la implementación, además de que puede afectar el rendimiento del cifrado.
4. **Expansión de datos.** Es extremadamente deseable que los datos cifrados no aumenten su tamaño respecto a los datos en claro.
5. **Propagación de error.** Descifrar datos que contienen errores de bit puede llevar a recuperar incorrectamente el texto en claro, además de propagar errores en los bloques pendientes por descifrar. Normalmente, el tamaño de bloque afecta el error de propagación.

   A continuación se listan algunos algoritmos de cifrado por bloques.

2.2.3. Redes Feistel

Consiste en un CIFRADO ITERATIVO que mapea bloques de texto en claro de tamaño $2t$bits (separados en bloques L_0, R_0 de tamaño t) a un texto cifrado R_r, L_r mediante un proceso de r RONDAS.

Donde cada subllave K_i se obtiene de la llave K.

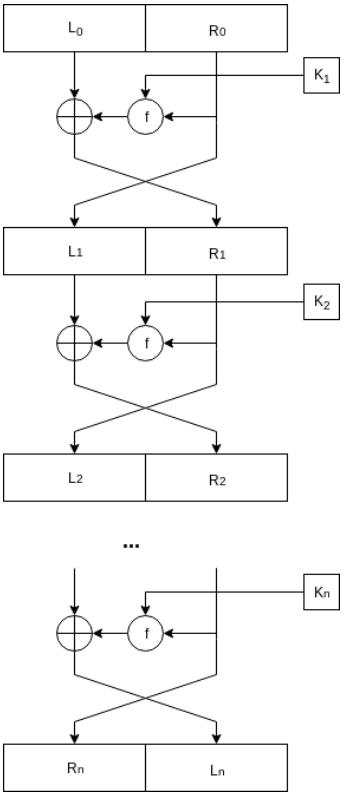


Figura 2.4: Diagrama genérico de una red Feistel.

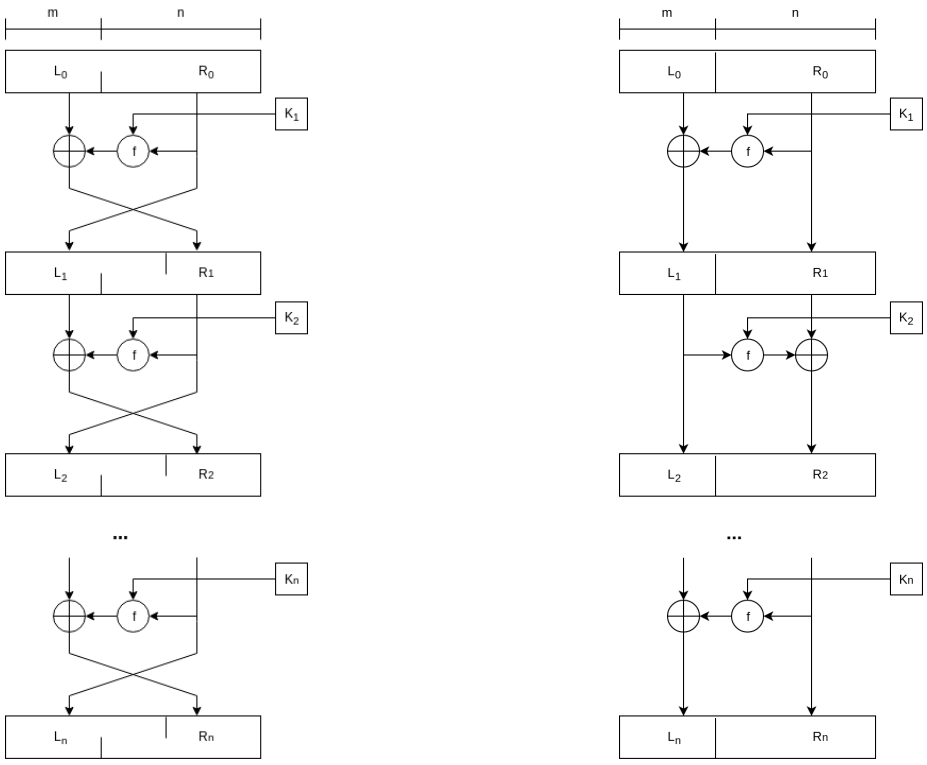
Normalmente el número de rondas r es mayor o igual a tres y par. Además, casi siempre intercambia el orden de los bloques de salida al revés en la última ronda: (R_r, L_r) en vez de (L_r, R_r).

El descifrado se realiza utilizando el mismo proceso de cifrado pero con las llaves en el orden inverso (comenzando con K_r hasta K_1).

Esta versión de las redes Feistel tiene como reestricción que la longitud del bloque a procesar debe ser par. Existen dos generalizaciones del esquema original que permiten procesar bloques de cualquier longitud: las redes Feistel desbalanceadas y las redes Feistel alternantes (figura 2.5) [6].

2.2.3.1. Redes Feistel desbalanceadas

Este tipo de redes (presentadas en [7]) permite que los bloques izquierdos y derechos sean de distintas longitudes (m y n, respectivamente). En la figura 2.5A se muestra un esquema del proceso, el cual es bastante similar al de la figura 2.4 con la única diferencia de que la función f debe cambiar la longitud de su entrada: de n a m. Si $m \leq n$, la red es pesada del lado de la fuente, y la función actúa como contracción (la entrada es más grande que la salida). Por otra parte, si $m \geq n$, la red es pesada del lado del objetivo, y la función actúa como una expansión (la entrada es más pequeña que la salida). El caso especial en el



(a) Redes desbalanceadas. (b) Redes alternantes.

Figura 2.5: Generalizaciones de las redes Feistel.

que $m = n$ es en el que la red está balanceada y corresponde al presentado originalmente (figura 2.4); es por esto que las redes Feistel desbalanceadas son consideradas una generalización del esquema inicial.

Para los esquemas pesados del lado de la fuente, la seguridad aumenta proporcionalmente al grado de desbalanceo. Por el lado contrario, en los esquemas pesados del lado del objetivo, entre más balanceada la red, mejor.

2.2.3.2. Redes Feistel alternantes

Un inconveniente de las redes desbalanceadas es el costo extra de hacer las particiones de los bloques intermedios. Las redes Feistel alternantes (figura 2.5B, presentadas en [8] y [9]) eliminan este inconveniente utilizando dos tipos de funciones, una contractora y la otra de expansión, en RONDAS alternas.

Es importante notar que las redes alternantes son también una generalización del esquema original, en la cuál la partición de los bloques es al centro, y se utliza una sola función.

2.2.4. Data Encryption Standard (DES)

Este es, probablemente, el cifrado simétrico por bloques más conocido; ya que en la década de los 70 estableció un precedente al ser el primer algoritmo a nivel comercial que publicó abiertamente sus especificaciones y detalles de implementación. Se encuentra definido en el estándar americano Federal Information Processing Standard (FIPS) 46-2.

DES es un cifrado Feistel que procesa bloques de $n = 64$ bits y produce bloques cifrados de la misma longitud. Aunque la llave es de 64 bits, 8 son de paridad, por lo que el tamaño *efectivo* de la llave es de 56 bits. Las 2^{56} llaves implementan, máximo, 2^{56} de las $2^{64}!$ posibles biyecciones en bloques de 64 bits.

Con la llave K se generan 16 subllaves K_i de 48 bits; una para cada ronda. En cada ronda se utilizan 8 *cajas-s* (mapeos de sustitución de 6 a 4 bits). La entrada de 64 bits es dividida por la mitad en L_0 y R_0. Cada ronda i va tomando las entradas L_{i-1} y R_{i-1} de la ronda anterior y produce salidas de 32 bits L_i y R_i mientras $1 \leq i \leq 16$ de la siguiente manera:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \\ &\quad donde \quad f(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i)) \end{aligned} \tag{2.4}$$

E se encarga de expandir R_{i-1} de 32 bits a 48, P es una permutación de 32 bits y S son las cajas-s.

El descifrado DES consiste en el mismo algoritmo de cifrado, con la misma llave K, pero utilizando las subllaves en orden inverso: $K_{16}, K_{15}, \ldots, K_1$.

2.2.4.1. Llaves débiles

Tomando en cuenta las siguientes definiciones

- Llave débil: una llave K tal que $E_K(E_K(M)) = M$ para toda x; en otras palabras, una llave débil permite que, al cifrar dos veces con la misma llave, se obtenga de nuevo el mensaje en claro.
- Llaves semidébiles: se tiene un par de llaves K_1, K_2 tal que $E_{K_1}(E_{K_2}(x)) = x$.

DES tiene cuatro llaves débiles y seis pares de llaves semidébiles. Las cuatro llaves débiles generan subllaves K_i iguales y, debido a que DES es un cifrado Feistel, el cifrado es autorreversible. O sea que al final se obtiene de nuevo el texto en claro, pues cifrar dos veces con la misma llave regresa la entrada original. Respecto a los pares semidébiles, el cifrado con una de las llaves del par es equivalente al descifrado con la otra (o viceversa).

```
1  entrada:  64 bits de texto en claro M = m_1 ... m_64;
2            llave de 64 bits K = k_1 ... k_64.
3  salida:   bloque de texto cifrado de 64 bits C = c_1 ... c_64.
4  inicio
5    Calcular 16 subllaves K_i de 48 bits partiendo de K.
6    Obtener (L_0, R_0) de la tabla de permutaciones iniciales IP(m_1 m_2 ... m_64)
7    para_todo i desde 1 hasta 16:
8      L_i = R_{i-1}
9      Obtener f(R_{i-1}, K_i):
10       a) Expandir R_{i-1} = r_1 r_2 ... r_32 de 32 a 48 bits
11         usando E: T <- E(R_{i-1}).
12       b) T' <- T ⊕ K_i. Donde T' es representado
13         como ocho cadenas de 6 bits cada una (B_1, ..., B_8).
14       c) T'' <- (S_1(B_1), S_2(B_2), ... S_8(B_8))
15       d) T''' <- P(T'')
16     R_i = L_{i-1} \oplus f(R_{i-1}, K_i)
17   fin
18   b_1 b_2 ... b_64 <- (R_16, L_16).
19   C <- IP^{-1}(b_1 b_2 ... b_64)
20 fin
```

Pseudocódigo 2.3: DES, cifrado.

```
entrada:    128 bits de texto en claro M; llave de n bits K.
salida:     bloque de texto cifrado de 64 bits C = c_1 ... c_64.
inicio
  Obtener las subllaves de 128 bits necesarias: una para cada ronda y una extra.
  Iniciar matriz de estado con el bloque en claro.
  Realizar AddRoundKey(matriz_estado, k_0)
  para_todo i desde 1 hasta num_rondas - 1:
    SubBytes(matriz_estado)
    ShiftRows(matriz_estado)
    MixColumns(matriz_estado)
    AddRoundKey(matriz_estado, k_i)
  fin
  SubBytes(matriz_estado)
  ShiftRows(matriz_estado)
  MixColumns(matriz_estado)
  AddRoundKey(matriz_estado, k_num_rondas)
  regresa matriz_estado
fin
```

Pseudocódigo 2.4: AES, cifrado.

2.2.5. Advanced Encryption Standard (AES)

Dado que el tamaño de bloque y la longitud de la llave de DES se volvieron muy pequeños para resistir los embates del progreso de la tecnología, el National Institute of Standards and Technology (NIST) comenzó la búsqueda de un nuevo cifrado estándar en 1997; este cifrado debía tener un tamaño de bloque de, al menos, 128 bits y soportar tres tamaños de llave: 128, 192 y 256 bits.

Después de pasar por un proceso de selección, la propuesta Rijndael fue seleccionada. Se le hicieron algunas modificaciones, pues Rijndael soporta combinaciones de llaves y bloques de longitud 128, 169, 192, 224 y 256; mientras que AES tiene fijo el tamaño de bloque y solo utiliza los tres tamaños de llave mencionados anteriormente. Dependiendo del tamaño de la llave, se tiene el número de RONDAS: 10 para las de 128 bits, 12 para las de 192 y 14 para las de 256.

El cifrado requiere de una matriz de 4×4 denominada matriz de estado.

Como todos los pasos realizados en las RONDAS son invertibles, el proceso de descifrado consiste en aplicar las funciones inversas a *SubBytes*, *ShiftRows*, *MixColumns* y *AddRoundKey* en el orden opuesto. Tanto el algoritmo como sus pasos están pensados con bytes. En el algoritmo Rijndael los bytes son considerados como elementos del campo finito $\mathbb{F}_{2^8}$ con 2^8 elementos; $\mathbb{F}_{2^8}$ es construido como una extensión del campo $\mathbb{F}_2$ con 2 elementos mediante el uso del polinomio irreducible $X^8 + X^4 + X^3 + X + 1$. Por lo tanto, las operaciones que se hagan a continuación de adición y el producto entre bytes significa

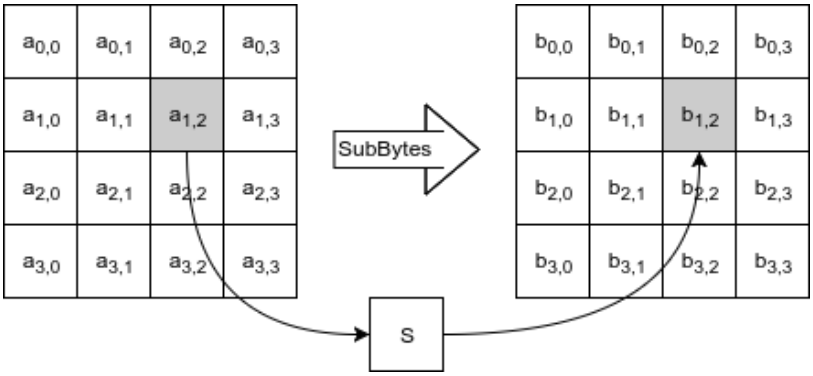


Figura 2.6: Diagrama de la operación *SubBytes*.

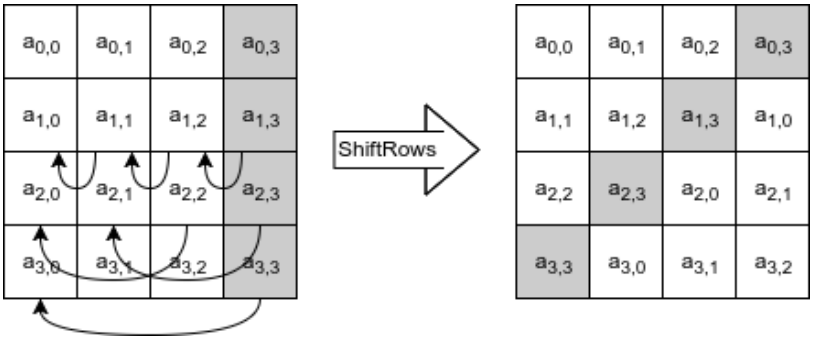


Figura 2.7: Diagrama de la operación $ShiftRows$.

sumarlos y multiplicarlos como elementos del campo $\mathbb{F}_{2^8}$.

2.2.5.1. SubBytes

Esta es la única transformación no lineal de Rijndael. Sustituye los bytes de la matriz de estado byte a byte al aplicar la función S_{RD} a cada elemento de la matriz. La función S_{RD} es también conocida como Caja-S y no depende de la llave. La misma caja es utilizada para los bytes en todas las posiciones.

2.2.5.2. ShiftRows

Esta transformación hace un corrimiento cíclico hacia la izquierda de las filas de la matriz de estado. Los desplazamientos son distintos para cada fila y dependen de la longitud del bloque (N_b).

2.2.5.3. MixColumns

Esta transformación opera en cada columna de la matriz de estado independientemente. Se considera una columna $a = (a_0, a_1, a_2, a_3)$ como el polinomio $a(X) = a_3X^3 + a_2X^2 + a_1X + a_0$. Entonces este paso

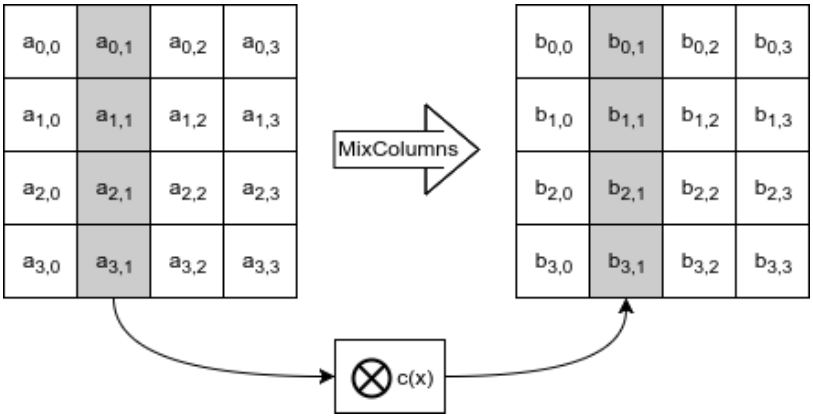


Figura 2.8: Diagrama de la operación $MixColumns$.

transforma una columna a al multiplicarla con el siguiente polinomio fijo:

$$c(X) = 03X^3 + 01X^2 + 01X + 02 \tag{2.5}$$

y se toma el residuo del producto módulo $X^4 + 1$:

$$a(X) \mapsto a(X) \cdot c(X) \quad \text{mód } (X^4 + 1) \tag{2.6}$$

2.2.5.4. AddRoundKey

Esta es la única operación que depende de la llave secreta k. Añade una llave de ronda para intervenir en el resultado de la matriz de estado. Las llaves de ronda son derivadas de la llave secreta k al aplicar el algoritmo de generación de llaves. Las llaves de ronda tienen la misma longitud que los bloques. Esta operación es simplemente una operación XOR bit a bit de la matriz de estado con la llave de ronda en turno. Para obtener el nuevo valor de la matriz de estado se realiza lo siguiente:

$$(matriz\_estado, k_i) \mapsto matriz\_estado \oplus k_i \tag{2.7}$$

Como se tiene una matriz_estado, la llave de ronda (k_i) también es representada como una matriz de bytes con 4 columnas y N_b columnas. Cada una de las N_b palabras de la llave de ronda corresponde a una columna. Entonces se realiza la operación XOR bit a bit sobre las entradas correspondientes de la matriz de estado y la matriz de la llave de ronda.

Esta operación, claro está, es invertible: basta con aplicar la misma operación con la misma llave para revertir el efecto.

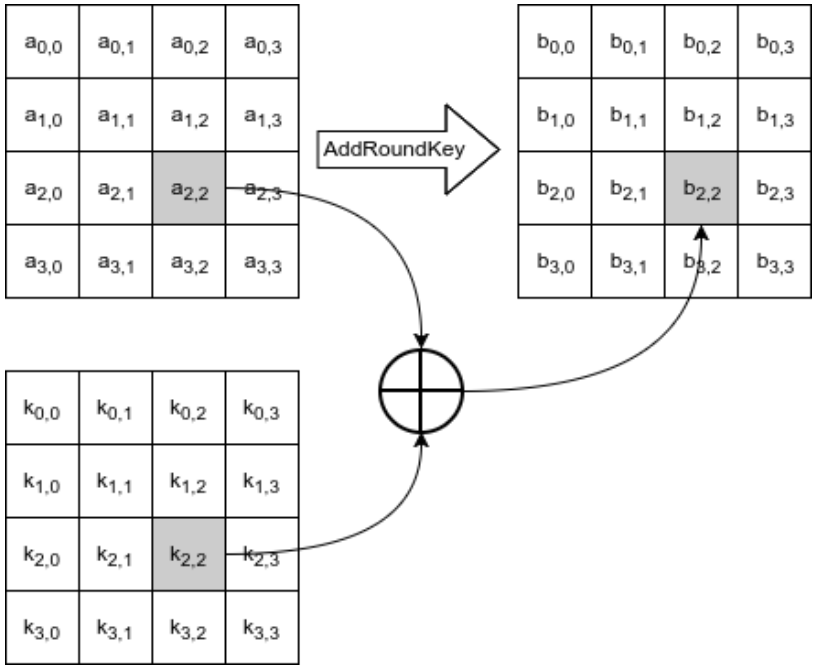


Figura 2.9: Diagrama de la operación *AddRoundKey*.

2.2.6. Fast Data Encipherment Algorithm (FEAL)

Es una familia de algoritmos que ha tenido una participación crítica en el desarrollo y refinamiento de varias técnicas del criptoanálisis, tales como el criptoanálisis lineal y diferencial. Fast Data Encipherment Algorithm (FEAL)-N mapea bloques de texto en claro de 64 bits a bloques de 64 bits de texto cifrado mediante una llave secreta de 64 bits. Es un cifrado Feistel de $n-$rondas parecido a DES, pero con una función f más simple.

FEAL fue diseñado para ser veloz y simple, especialmente para microprocesadores de 8 bits: usa operaciones orientadas a bytes, evita el uso de permutaciones de bit y tablas de consulta. La versión inicial de cuatro rondas (FEAL-4), propuesto como una alternativa rápida a DES, fue encontrado mucho más inseguro de lo planeado; por lo que se propuso realizar más rondas (FEAL-16 y FEAL-32) para compensar y ofrecer un nivel de seguridad parecido a DES; sin embargo, el rendimiento se ve fuertemente afectado mientras el número de rondas aumenta; y, mientras DES puede mejorar su velocidad con tablas de consulta, resulta más complicado para FEAL.

Para descifrar se utiliza el mismo algoritmo, con la misma llave K y el texto cifrado $C = (R_8, L_8)$ se utiliza como la entrada M; sin embargo, la generación de llaves se hace al revés: las subllaves $((K_{12}, K_{13}), (K_{14}, K_{15}))$ se utilizan para la $\oplus$ inicial, las $((K_8, K_9), (K_{10}, K_{11}))$ para la $\oplus$ final y en las rondas se utiliza de la subllave K_7 a la K_0.

FEAL con una llave de 64 bits puede ser generalizado a un número n de rondas rondas con n par,

```
1  entrada:    64 bits de texto en claro M = m_1 ... m_64;
2              llave de 64 bits K = k_1 ... k_64.
3  salida:     bloque de texto cifrado de 64 bits C = c_1 ... c_64.
4  inicio
5    Calcular 16 subllaves de 16 bits para K.
6    Definir M_L = m_1 ... m_32; M_R = m_33 ... m_64.
7    (L_0, R_0) ← (M_L, M_R) ⊕ ((K_8, K_9), (K_10, K_11))
8    R_0 ← R_0 ⊕ L_0.
9    para_todo i desde 1 hasta 8:
10     L_i ← R_{i-1}
11     R_i ← L_{i-1} ⊕ f(R_{i-1}, K_{i-1})
12   fin
13   L_8 ← L_8 ⊕ R_8
14   (R_8, L_8) ← (R_8, L_8) ⊕ ((K_12, K_13), (K_14, K_15))
15   C ← (R_8, L_8).
16 fin
```

Pseudocódigo 2.5: FEAL-8, cifrado.

aunque se recomienda $n = 2^x$.

2.2.7. International Data Encryption Algorithm (IDEA)

Cifra bloques de 64 bits utilizando una llave de 128 bits. Este cifrado está basado en una generalización de la estructura Feistel y consiste en 8 RONDAS idénticas seguidas por una transformación. Cada ronda r utiliza 6 subllaves $K_i^{(r)}$ $(1 \leq i \leq 6)$ de 16 bits que se encargan de transformar una entrada X de 64 bits en una salida de cuatro bloques de 16-bits, que son utilizados como entrada en la siguiente ronda. La salida de la ronda 8 tiene como entrada la transformación de salida que, al emplear cuatro llaves adicionales $K_i^{(9)}$ $(1 \leq i \leq 4)$, produce los datos cifrados $Y = (Y_1, Y_2, Y_3, Y_4)$.

El descifrado se realiza con el mismo algoritmo de cifrado, pero utilizando como entrada los datos cifrados Y como entrada M. Se usa la misma llave K; aunque las subllaves sufren una modificación al ser generadas, pues se utiliza una tabla y se realizan las operaciones contrarias (inverso de la adición y el inverso del producto).

Descartando los ataques a las llaves débiles, no hay un mejor ataque publicado para el INTERNATIONAL DATA ENCRYPTION ALGORITHM (IDEA) de 8 RONDAS que el de la búsqueda exhaustiva en el espacio de llave. Por lo que la seguridad está ligada a la creciente debilidad de su tamaño de bloque relativamente pequeño.

```
1  entrada:   64-bits de datos en claro M = m_1 ... m_64;
2             llave de 128-bits K = k_1 ... k_128.
3  salida:    bloque cifrado de 64-bits Y = (Y_1, Y_2, Y_3, Y_4).
4  inicio
5    Calcular las subllaves K_1^(r),...,K_6^(r) para las rondas 1 ≤ r ≤ 8 y K_1^(9),...,K_4^(9)
6    para la transformación de salida.
7    (X_1, X_2, X_3, X_4) ← (m_1 ... m_16, m_17 ... m_32, m_33 ... m_48, m_49 ... m_64)
8        donde X_i almacena 16 bits.
9    para_todo r desde 1 hasta 8:
10     a) X_1 ← X_1 × K_1^(r)  mód 2^16 + 1
11        X_4 ← X_4 × K_4^(r)  mód 2^16 + 1
12        X_2 ← X_2 + K_2^(r)  mód 2^16
13        X_3 ← X_3 + K_3^(r)  mód 2^16
14     b) t_0 ← K_5^(r) × (X_1 ⊕ X_3)  mód 2^16 + 1
15        t_1 ← K_6^(r) × (t_0 + (X_2 ⊕ X_4))  mód 2^16 + 1
16        t_2 ← t_0 + t_1  mód 2^16
17     c) X_1 ← X_1 ⊕ t_1
18        X_4 ← X_4 ⊕ t_2
19        a ← X_2 ⊕ t_2
20        X_2 ← X_3 ⊕ t_1
21        X_3 ← a
22   fin
23   Realizar la transformación de salida:
24     Y_1 ← X_1 × K_1^(9)  mód 2^16 + 1
25     Y_4 ← X_4 × K_4^(9)  mód 2^16 + 1
26     Y_2 ← X_3 + K_2^(9)  mód 2^16
27     Y_3 ← X_2 + K_3^(9)  mód 2^16
28 fin
```

Pseudocódigo 2.6: IDEA, cifrado.

```
1  entrada: r, 6 ≤ r ≤ 10; 64−bits de datos en claro M = m_1 ... m_64; K = k_1 ... k_64.
2  salida: bloque cifrado de 64−bits Y = (Y_1, ..., Y_8).
3  inicio
4    Calcular las subllaves K_1, ..., K_{2r+1}
5    (X_1, X_2, ... X_8) ← (m_1 ... m_8, m_9 ... m_16, ..., m_57 ... m_64)
6    para_todo i desde 1 hasta r:
7      a) Para j = 1,4,5,8 : X_j ← X_j ⊕ K_{2i−1}[j]
8         Para j = 2,3,6,7 : X_j ← X_j + K_{2i−1}[j] mod 2^8
9      b) Para j = 1,4,5,8 : X_j ← S[X_j]
10        Para j = 2,3,6,7 : X_j ← S_inversa X_j
11     c) Para j = 1,4,5,8 : X_j ← X_j + K_{2i}[j] mod 2^8
12        Para j = 2,3,6,7 : X_j ← X_j ⊕ K_{2i}[j]
13     d) Para j = 1,3,5,7 : (X_j, X_{j+1}) ← f(X_j, X_{j+1}).
14     e) (Y_1, Y_2) ← f(X_1, X_3), (Y_3, Y_4) ← f(X_5, X_7),
15        (Y_5, Y_6) ← f(X_2, X_4), (Y_7, Y_8) ← f(X_6, X_8).
16        Para j desde 1 hasta 8: X_j ← Y_j
17     f) (Y_1, Y_2) ← f(X_1, X_3), (Y_3, Y_4) ← f(X_5, X_7),
18        (Y_5, Y_6) ← f(X_2, X_4), (Y_7, Y_8) ← f(X_6, X_8).
19        Para j desde 1 hasta 8: X_j ← Y_j.
20   fin
21   Para j = 1,4,5,8 : Y_j ← X_j ⊕ K_{2r+1}[j].
22   Para j = 2,3,6,7 : Y_j ← X_j + K_{2r+1}[j] mód 2^8.
23 fin
```

Pseudocódigo 2.7: SAFER K-64, cifrado.

2.2.8. Secure And Fast Encryption Routine (SAFER)

El cifrado Secure And Fast Encryption Routine (SAFER) K-64 es un cifrado por bloques de 64 bits iterativo. Consiste en r RONDAS idénticas seguidas por una transformación. Originalmente se recomendaban 6 RONDAS seguidas, sin embargo, ahora se utiliza una generación de claves ligeramente modificada y el uso de 8 RONDAS (máximo 10). Ambas generaciones de llaves expanden la llave de 64 bits en $2r + 1$ subllaves, cada una de 64 bits (dos por cada ronda y una más para la transformación de salida).

Este cifrado consiste completamente en operaciones de bytes, por lo que es adecuado para procesadores con tamaños de palabra pequeños, como los chips de tarjetas.

Para descifrar, se utiliza la misma llave K y las subllaves K_i que fueron utilizadas al cifrar. Cada paso del cifrado se hace en orden inverso, del último al primero; comenzando con una transformación de entrada utilizando la llave K_{2r+1} para deshacer la transformación de salida, se sigue con las RONDAS de descifrado utilizando las llaves de K_{2r} a K_1, invirtiendo los pasos cada ronda.

```
entrada: 2w−bits de datos en claro M = (A, B); r;
    llave K = K[0]...K[b − 1]
salida: 2w−bits de datos cifrados C.
inicio
  Calcular 2r + 2 subllaves K_0, ..., K_{2r+1}
  A ← A + K_0 mód 2^w, B ← B + K_1 mód 2^w
  para_todo i desde 1 hasta r:
    A ← ((A ⊕ B) ↩ B) + K_{2i} mód 2^w
    B ← ((B ⊕ A) ↩ A) + K_{2i+1} mód 2^w
  fin
  Regresar C ← (A, B)
fin
```

Pseudocódigo 2.8: RC5, cifrado.

```
entrada: 2w−bits de datos cifrados C = (A, B); r;
    llave K = K[0]...K[b − 1]
salida: 2w−bits de datos en claro M.
inicio
  Calcular 2r + 2 subllaves K_0, ..., K_{2r+1}
  A ← A + K_0 mód 2^w, B ← B + K_1 mód 2^w
  Para i desde r hasta 1:
    B ← ((B − K_{2i+1} mód 2^w) ↪ A) ⊕ A
    A ← ((A − K_{2i} mód 2^w) ↪ B) ⊕ B
  fin
  Regresar M ← (A − K_0 mód 2^w, B − K_1 mód 2^w)
fin
```

Pseudocódigo 2.9: RC5, descifrado.

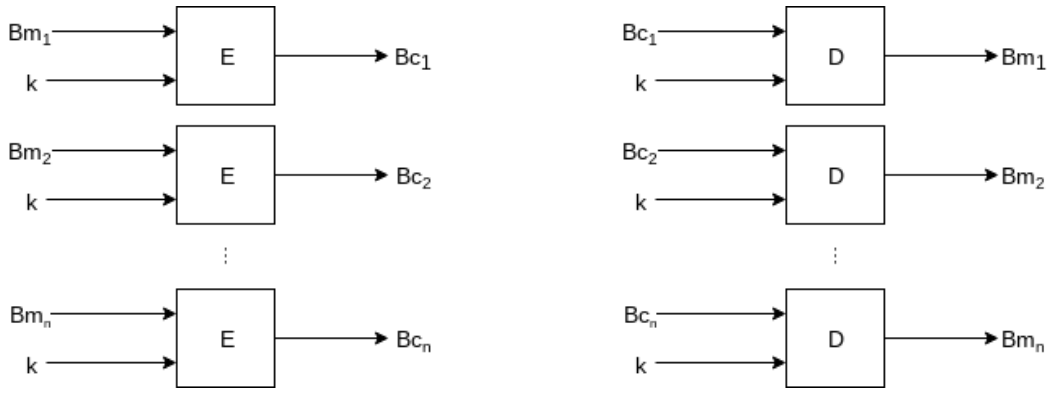
2.2.9. RC5

Este cifrado por bloques tiene una arquitectura orientada a palabras (ya sea $w = 16, 32, 64$bits) y tiene una descripción muy compacta adecuada tanto para hardware como para software. Tanto la longitud b de la llave y el número de RONDAS r es variable; aunque se recomiendan 12 RONDAS para 32 bits y 16 para cuando se tienen palabras de 64.

Para descifrar, RC5 utiliza el siguiente algoritmo.

2.2.10. Modos de operación

La información que aquí se presenta se puede consultar a mayor detalle en [10] y [1].



(a) Cifrado. (b) Descifrado.

Figura 2.10: Modo de operación ECB.

Por sí solos, los cifrados por bloques solamente permiten el cifrado y descifrado de bloques de información de tamaño fijo; donde, en la mayoría de los casos, los bloques son de menos de 256 bits, lo cual es equivalente a alrededor de 8 caracteres. Es fácil darse cuenta de que esta restricción no es ningún tema menor: en la gran mayoría de las aplicaciones, la longitud de lo que se quiere ocultar es arbitraria.

Los modos de operación permiten extender la funcionalidad de los cifrados por bloques para poder aplicarlos a información de tamaño irrestricto: reciben el texto original (de tamaño arbitrario) y lo cifran, ocupando en el proceso un cifrado por bloques.

Un primer enfoque (y quizás el más intuitivo) es partir el mensaje original en bloques del tamaño requerido y después aplicar el algoritmo a cada bloque por separado; en caso de que la longitud del mensaje no sea múltiplo del tamaño de bloque, se puede agregar información extra al último bloque para completar el tamaño requerido. Este es, de hecho, el primero de los modos que se presentan a continuación, el Electronic Codebook (ECB); su uso no es recomendado, pues es muy inseguro cuando el mensaje original es simétrico a nivel de bloque. También se enlistan otros tres modos, los cuales junto con ECB, son los más comunes.

2.2.10.1. *Electronic Codebook* (ECB)

La figura 2.10 muestra un diagrama esquemático de este modo de operación. El algoritmo recibe a la entrada una llave y un mensaje de longitud arbitraria: la llave se pasa sin ninguna modificación a cada función del cifrado por bloques; el mensaje se debe de partir en bloques ($M = Bm_1||Bm_2||\ldots||Bm_n$).

2.2.10.2. *Cipher-block Chaining* (CBC)

En Cipher-block Chaining (CBC) la salida del bloque cifrador uno se introduce (junto con el siguiente bloque del mensaje) en el bloque cifrador dos, y así en sucesivo. Para poder replicar este comportamiento

```
1  entrada: llave k; bloques de mensaje Bm_1, Bm_2 ... Bm_n.
2  salida:  bloques de mensaje cifrado Bc_1, Bc_2 ... Bc_n.
3  inicio
4    para_todo Bm
5      Bc_i ← E_k(Bm_i)
6    fin
7    regresar Bc
8  fin
```

Pseudocódigo 2.10: Modo de operación ECB, cifrado.

```
1  entrada: llave k; bloques de mensaje cifrado Bc_1, Bc_2 ... Bc_n.
2  salida:  bloques de mensaje original B_1, B_2 ... B_n.
3  inicio
4    para_todo Bc
5      Bm_i ← D_k(Bc_i)
6    fin
7    regresar Bm
8  fin
```

Pseudocódigo 2.11: Modo de operación ECB, descifrado.

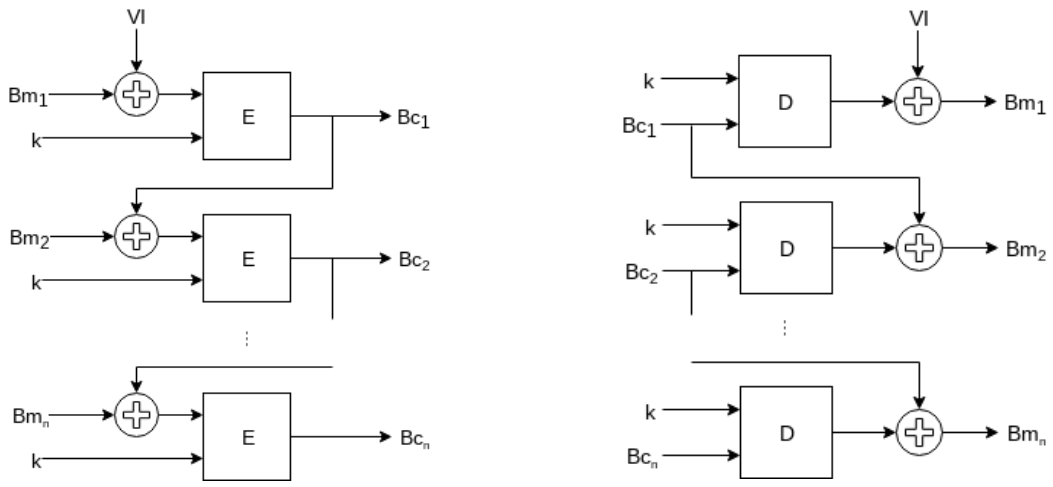
en todos los bloque cifradores, este modo de operación necesita un argumento extra a la entrada: un vector de inicialización. De esta manera la salida del bloque i depende de todos los bloques anteriores; esto incrementa la seguridad con respecto a ECB.

En la figura 2.11 se muestran los diagramas esquemáticos para cifrar y descifrar; en los pseudocódigos 2.12 y 2.13 se muestran unos de los posibles algoritmos a seguir. Es importante notar que mientras que el proceso de cifrado debe ser forzosamente secuencial (por la dependencias entre salidas), el proceso de descifrado puede ser ejecutado en paralelo.

2.2.10.3. *Cipher Feedback* (CFB)

Al igual que la operación de cifrado de CBC, ambas operaciones de Cipher Feedback (CFB) (cifrado y descifrado) están encadenadas bloque a bloque, por lo que son de naturaleza secuencial. En este caso, lo que se cifra en el primer paso es el vector de inicialización; la salida de esto se opera con un `xor` sobre el primer bloque de texto en claro, para obtener el primer bloque cifrado (figura 2.12).

Esta distribución presenta varias ventajas con respecto a CBC: las operaciones de cifrado y descifrado son sumamente similares, lo que permite ser implementadas por un solo algoritmo (pseudocódigo 2.14); tanto para cifrar como para descifrar solamente se ocupa la operación de cifrado del algoritmo a bloques



(a) Cifrado. (b) Descifrado.

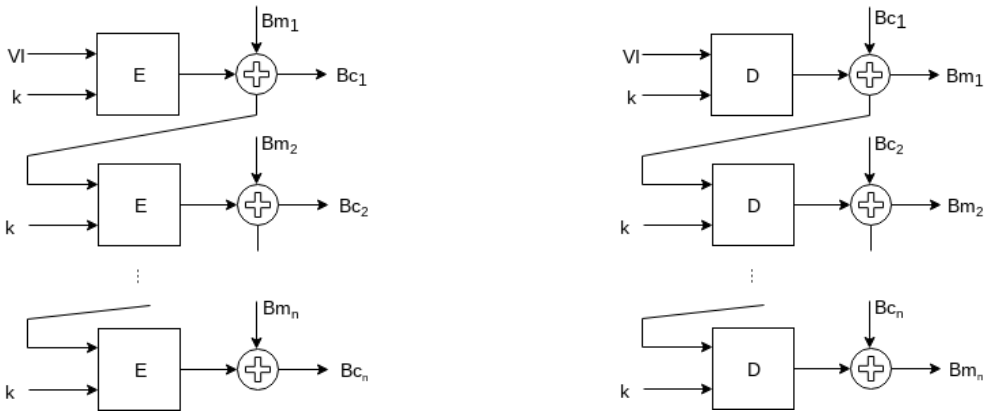
Figura 2.11: Modo de operación CBC.

```
1  entrada: llave k; vector de inicialización VI;
2           bloques de mensaje Bm_1, Bm_2 ... Bm_n.
3  salida:  bloques de mensaje cifrado Bc_1, Bc_2 ... Bc_n.
4  inicio
5    Bc_0 ← VI                    // El vector de inicialización
6    para_todo Bm                 // entra al primer bloque.
7      Bc_i ← E(k, Bm_i ⊕ Bc_{i-1})
8    fin
9    regresar Bc
10 fin
```

Pseudocódigo 2.12: Modo de operación CBC, cifrado.

```
1  entrada: llave k; vector de inicialización VI;
2           bloques de mensaje cifrado Bc_1, Bc_2 ... Bc_n.
3  salida:  bloques de mensaje original Bm_1, Bm_2 ... Bm_n.
4  inicio
5    Bc_0 ← VI
6    para_todo Bc
7      Bm_i ← D_k(Bc_i) ⊕ Bc_{i-1}
8    fin
9    regresar Bm
10 fin
```

Pseudocódigo 2.13: Modo de operación CBC, descifrado.



(a) Cifrado. (b) Descifrado.

Figura 2.12: Modo de operación CFB.

```
1  entrada: llave k; vector de inicialización VI;
2           bloques de mensaje (cifrado o descifrado) Bm_1, Bm_2 ... Bm_n.
3  salida:  bloques de mensaje (cifrado o descifrado) Bc_1, Bc_2 ... Bc_n.
4  inicio
5    Bc_0 ← VI
6    para_todo Bm
7      Bc_i ← C_k(Bc_{i-1}) ⊕ Bm_i
8    fin
9    regresar Bc
10 fin
```

Pseudocódigo 2.14: Modo de operación CFB(cifrado y descifrado).

subyacente. Estas ventajas se deben principalmente a las propiedades de la operación `xor` (ecuación 2.8).

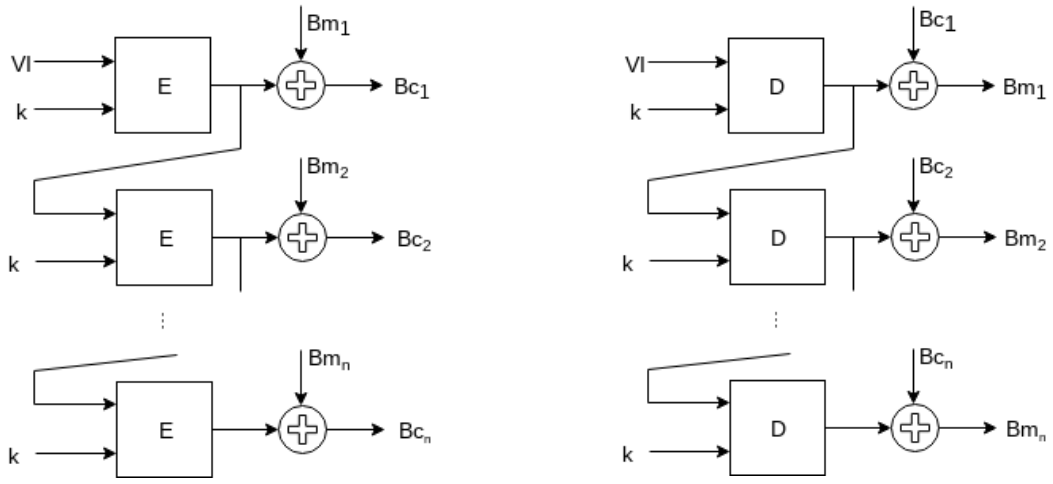
$$A \oplus B = C \quad \Rightarrow \quad A = B \oplus C \tag{2.8}$$

2.2.10.4. *Output Feedback* (OFB)

Este modo es muy similar al anterior (CFB), salvo que la retroalimentación va directamente de la salida del cifrador a bloques. De esta forma, nada que tenga que ver con el texto en claro llega al cifrado a bloques; este solamente se la pasa cifrando una y otra vez el vector de inicialización.

2.2.10.5. *Counter Mode* (CTR)

Este opera de manera un tanto distinta con respecto a los anteriores: toma a la entrada un vector de inicialización y en cada iteración lo incrementa y lo cifra. El resultado se obtiene combinando el VI



(a) Cifrado. (b) Descifrado.

Figura 2.13: Modo de operación Output Feedback (OFB).

```
1  entrada: llave k; vector de inicialización VI;
2           bloques de mensaje (cifrado o descifrado) Bm_1, Bm_2 ... Bm_n.
3  salida:  bloques de mensaje (cifrado o descifrado) Bc_1, Bc_2 ... Bc_n.
4  inicio
5    auxiliar ← VI
6    para_todo Bm
7      auxiliar ← E_k(auxiliar)
8      Bc_i ← auxiliar ⊕ Bm_i
9    fin
10   regresar Bc
11 fin
```

Pseudocódigo 2.15: Modo de operación OFB(cifrado y descifrado).

```
1  entrada: llave k; vector de inicialización VI;
2           bloques de mensaje (cifrado o descifrado) Bm_1, Bm_2 ... Bm_n.
3  salida:  bloques de mensaje (cifrado o descifrado) Bc_1, Bc_2 ... Bc_n.
4  inicio
5    para_todo Bm
6      Bc_i ← E_k((VI + i) mód 2^n) ⊕ Bm_i
7    fin
8    regresar Bc
9  fin
```

Pseudocódigo 2.16: Modo de operación Counter Mode (CTR)(cifrado y descifrado).

cifrado con el bloque de texto cifrado (mediante una operación `xor`). El proceso de cifrado y descifrado se detallan en el pseudocódigo 2.16.

En términos de eficiencia, el CTR es mejor que CBC, CFB o OFB, ya que sus operaciones (ambas) se pueden hacer en paralelo. La implementación es prácticamente la misma para el cifrado y descifrado (solamente se ocupa el cifrado del algoritmo por bloques subyacente).

2.3. Cifrados de flujo

La información de esta sección (junto con las subsecciones contenidas) puede ser encontrada con mayor detalle en [1], [11] y [12].

A diferencia de los cifrados de bloque, que trabajan sobre grupos enteros de bits a la vez, los cifrados de flujo trabajan sobre bits individuales, cifrándolos uno por uno. Una manera de verlos es como cifrados por bloques con un tamaño de bloque igual a 1.

Un cifrado de flujo aplica transformaciones de acuerdo a un flujo de llave: una secuencia de símbolos pertenecientes al espacio de llaves. El flujo de llave puede ser generado tanto de manera aleatoria, como por un algoritmo pseudoaleatorio que reciba a la entrada, o bien una semilla, o bien una semilla y algunos bits del texto cifrado.

Entre las ventajas de los cifrados de flujo sobre los cifrados de bloque se encuentra el hecho de que son más rápidos en hardware y más útiles cuando el buffer es limitado o se necesita procesar la información al momento de llegada. La propagación de los errores es limitada o nula, por lo que también son más apropiados en casos en los que hay probabilidades altas de errores en la transmisión.

Los cifrados de bloques funcionan sin ninguna clase de memoria (por sí solos); en contraste, la función de cifrado de un cifrado de flujo puede variar mientras se procesa el texto en claro, por lo cuál tienen un mecanismo de memoria asociado. Otra denominación para estos cifrados es *de estado*, por que la salida no depende solamente del texto en claro y de la llave, sino que también depende del estado actual.

2.3.1. Clasificación

Una clasificación común es en *síncronos* y en *autosincronizables*. A continuación se describen a grandes rasgos ambos modelos.

2.3.1.1. Síncronos

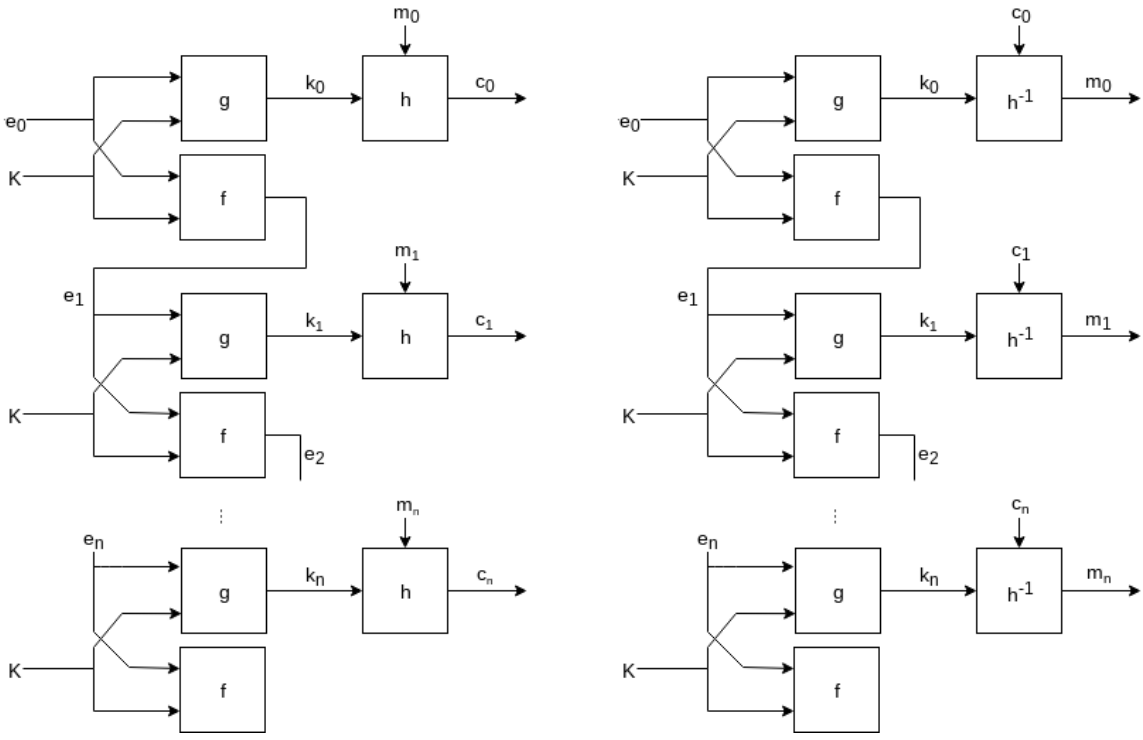
Un cifrado de flujo síncrono es aquel en el que el flujo de la llave es generado de manera independiente del texto en claro y del texto cifrado. Se puede definir un modelo general con las siguientes tres ecuaciones.

$$e_{i+1} = f(e_i, K) \tag{2.9}$$

$$k_i = g(e_i, K) \tag{2.10}$$

$$c_i = h(k_i, m_i) \tag{2.11}$$

La letra e representa el estado del cifrado, K es la llave, k es la salida del flujo de llave, c es el texto cifrado y m es el texto en claro. La función de la ecuación 2.9 (f) es la que describe el cambio de estado; este se determina a partir del estado actual y de la llave. En la ecuación 2.10 se describe la acción del flujo de



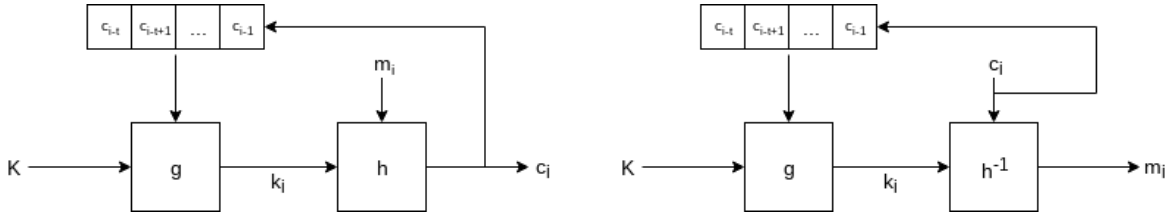
(a) Cifrado. (b) Descifrado.

Figura 2.14: Esquema general de un cifrado de flujo síncrono.

llave (g): para determinar el próximo símbolo se emplea solamente el estado actual y la llave. La tercera ecuación (2.11, h) describe la acción de combinar el flujo de la llave con el mensaje, y así obtener el texto cifrado.

En la figura 2.14 se describe de manera gráfica las operaciones de cifrado y descifrado; estas guardan muchas similitudes con el modo de operación OFB (sección 2.2.10.4), con la única excepción de que este trabaja con bloques del tamaño del cifrado subyacente. En otras palabras, si se definiera el tamaño del bloque (y en consecuencia el tamaño del VECTOR DE INICIALIZACIÓN) como 1, entonces OFB sería un cifrado de flujo síncrono.

El nombre de esta categoría proviene del hecho de que ambos entes del proceso comunicativo (emisor y receptor) deben encontrarse sincronizados (usar la misma llave y encontrarse en la misma posición) para que la comunicación tenga éxito: si se insertan dígitos extras al mensaje cifrado, la sincronización se pierde. Los cifrados de flujo síncronos no tienen propagación de error: aunque ciertos bits sean modificados (pero no borrados) durante su transmisión, el resto del mensaje sigue siendo descifrable.



(a) Cifrado. (b) Descifrado.

Figura 2.15: Esquema general de un cifrado de flujo autosincronizable.

2.3.1.2. Autosincronizables

En esta clasificación se engloban a aquellos cifrados cuyo flujo de llave es resultado de la propia llave original y de cierto número previo de dígitos cifrados. Las ecuaciones que describen su comportamiento son las siguientes.

$$e_{i+1} = (c_{i-t}, c_{i-t+1}, \ldots, c_{i-1}) \tag{2.12}$$

$$k_i = g(e_i, K) \tag{2.13}$$

$$c_i = h(k_i, m_i) \tag{2.14}$$

La notación es la misma que en las ecuaciones 2.9, 2.10 y 2.11. En este caso, el próximo estado depende de t (el tamaño de la ventana) dígitos cifrados anteriormente. En la figura 2.15 se describe de manera gráfica el proceso de cifrado y descifrado.

En una antítesis de la categoría anterior, el nombre de esta indica que no es necesario que el emisor y el receptor estén sincronizados: si se llegan a perder bits en la transmisión, el esquema es capaz de autosincronizarse, pues el flujo de la llave depende de cierto número de bits anteriores. A esta categoría también se le conoce como «asíncrona».

La propagación de los errores depende del tamaño de ventana (el número t de bits previos utilizados para calcular la próxima llave), si se modifica un bit, entonces los próximos t serán incorrectos.

2.3.2. RC4

RC4 es un cifrado de flujo diseñado por Ron L. Rivest en 1987 para la empresa RSA. Es usado en varios protocolos de seguridad comunes: Secure Sockets Layer (SSL)/Transport Layer Security (TLS), Wired Equivalent Privacy (WEP) y WiFi Protected Access (WPA); los dos últimos son parte del estándar Institute of Electrical and Electronic Engineers (IEEE) 802.11 para comunicaciones Local Area Network (LAN) inalámbricas. RC4 se mantenía como secreto de compañía hasta que, en septiembre de 1994, fue filtrado de forma anónima en Internet.

En el pseudocódigo 2.17 se describe el proceso de cifrado del algoritmo. S es un vector de estado; T

```
1  entrada: llave k; mensaje original m_1, m_2 ... m_n.
2  salida:  mensaje cifrado c_1, c_2 ... c_n.
3  inicio
4    /* Inicialización */
5    para_todo i entre 0 y 255:
6      S[i] ← i
7      T[i] ← K[i  mód  longitud_de_llave]
8    fin
9
10   /* Permutación inicial */
11   j ← 0
12   para_todo i entre 0 y 255:
13     j = (j + S[i] + T[i])  mód  256
14     intercambiar(S[i], S[j])
15   fin
16
17   /* Proceso de cifrado */
18   i, j ← 0
19   para_todo m:
20     i ← (i + 1)  mód  256
21     j ← (j + S[i])  mód  256
22     intercambiar(S[i], S[j])
23     k ← S[(S[i] + S[j])  mód  256]
24     c ← m ⊕ k
25   fin
26   regresar c
27 fin
```

Pseudocódigo 2.17: Proceso de cifrado de RC4.

es un vector temporal.

Se han hecho varias publicaciones que analizan métodos para atacar RC4, ninguna de las cuales presenta algo práctico cuando se utiliza una llave mayor a 128 bits. Sin embargo, en [13] se reporta un problema más serio sobre la implementación que se hace de RC4 en el protocolo WEP; este problema en particular no ha demostrado afectar a otras aplicaciones que usan RC4.

2.3.3. El proyecto eSTREAM

La información aquí expuesta puede ser consultada a mayor detalle en [14], [15] y [16].

El proyecto eSTREAM fue un esfuerzo de la comunidad European Network of Excellence in

Perfil 1	**Perfil 2**
HC-128 Rabbit Salsa20/12 Sosemanuk	Grain v1 MICKEY v2 Trivium

Tabla 2.1: Finalistas del proyecto eSTREAM

Cryptology (ECRYPT) para promover el diseño de cifrados de flujo eficientes y compactos. Como resultado, se publicó un portafolio en abril de 2008, el cual ha estado bajo continuas actualizaciones desde entonces; actualmente cuenta con siete algoritmos (tabla 2.1).

El portafolio se divide en dos perfiles: el primero contiene algoritmos adecuados para aplicaciones de software con requerimientos de rapidez de procesamiento muy altos; el segundo se enfoca en aplicaciones de hardware con pocos recursos disponibles.

El proyecto se inició después de que Adi Shamir se preguntara si realmente había necesidad de los cifrados de flujo, en una conferencia de RSA en 2004. El principal argumento a favor fue que, para la gran mayoría de los casos, el uso de AES (sección 2.2.5) con una configuración de flujo es una solución adecuada. Este último punto de vista iba generalmente acompañado de la creencia de que era imposible diseñar cifrados de flujo seguros (un proyecto anterior similar, New European Schemes for Signatures, Integrity and Encryption (NESSIE), terminó sin resultados después de todos los criptoanálisis hechos). Por otra parte, como argumentos en contra, Shamir identificó dos áreas en las que los cifrados de flujo ofrecen ventajas respecto a los cifrados por bloques:

1. Cuando se requieren tiempos de procesamiento excepcionalmente rápidos (perfil 1).
2. Cuando los recursos disponibles son muy pocos (perfil 2).

Las palabras de Shamir fueron ampliamente difundidas, y más tarde en ese mismo año, ECRYPT lanzó el proyecto eSTREAM, cuyo principal objetivo fue ampliar el conocimiento sobre el análisis y el diseño de cifrados de flujo. Después de un periodo de estudio, se lanzó una convocatoria que generó un interés considerable: antes del 29 de abril de 2005 (la fecha límite) se recibieron 34 propuestas; algunas de las cuales intentaban cumplir con los dos perfiles a la vez (lamentablemente no sobrevivieron mucho tiempo).

2.4. Funciones hash

La información presentada a continuación puede consultarse con más profundidad en las siguientes referencias [1], [2], [17], [18].

Se refiere al conjunto de funciones computacionalmente eficientes que mapean cadenas binarias de una longitud arbitraria a cadenas binarias de una longitud fija, llamadas valores hash.

Matemáticamente, una función hash es una función

$$\begin{aligned} h : \{0,1\}^* &\longrightarrow \{0,1\}^n \\ m &\longmapsto h(m) \end{aligned} \tag{2.15}$$

La longitud de n suele ser entre 128 y 512 bits. Las funciones hash h tienen las siguientes propiedades:

1. Compresión: h mapea una entrada x (cuya longitud finita es arbitraria) a una salida $h(x)$ de longitud fija n.
2. Facilidad de cómputo: dada x y h, $h(x)$ es calculada ya sea sin necesitar mucho espacio, tiempo de cómputo, o requiere pocas operaciones, etcétera.

De manera general, las funciones hash se pueden dividir en dos categorías: las que no utilizan llave y su único parámetro es la entrada x, y las que necesitan una llave secreta k y la entrada x.

Sea una función hash sin llave h con entradas x, x' y salidas y y y', respectivamente. A continuación se listan algunas de las propiedades que puede tener:

1. Resistencia de PREIMAGEN: no es computacionalmente factible para una salida específica y encontrar una entrada x' que dé como resultado el mismo valor hash $h(x') = y$ si no se conoce x. Esta propiedad también es llamada *de un sentido*.
2. Resistencia de segunda PREIMAGEN: no es computacionalmente factible encontrar una segunda entrada x' que tenga la misma salida que una entrada específica x: $x \neq x'$ tal que $h(x) = h(x')$. Esta propiedad también es conocida como *de débil resistencia a colisiones*.
3. Resistencia a las colisiones: no es computacionalmente factible encontrar dos entradas distintas x, x' que lleven al mismo valor hash, o sea, $h(x) = h(x')$. A diferencia de la anterior, la selección de ambas entradas no está restringida. Esta propiedad también es conocida como *de gran resistencia a colisiones*.

Una función hash h que cumple con las propiedades de resistencia de PREIMAGEN y resistencia de segunda PREIMAGEN es conocida como una función hash de un solo sentido o ONE-WAY HASH FUNCTION (OWHF). Las que cumplen con la resistencia de segunda PREIMAGEN y resistencia a las colisiones son

conocidas como funciones hash resistentes a colisiones o Collision-Resistant Hash Function (CRHF). Aunque casi siempre las funciones CRHF cumplen con la resistencia de preimagen, no es obligatorio que lo hagan.

Algunos ejemplos de las funciones OWHF son el Secure Hash Algorithm (SHA)-1 y el Message Digest-5 (MD5). En los esquemas de firma electrónica, se obtiene el valor hash del mensaje ($h(m)$) y se pone en el lugar de la firma. Los valores hash también son utilizados para revisar la integridad de las llaves públicas y, al utilizarse con una llave secreta, las funciones criptográficas hash se convierten en códigos de autenticación de mensaje (MAC, por sus siglas en inglés), una de las herramientas más utilizadas en protocolos como SSL e IPSec para revisar la integridad de un mensaje y autenticar al remitente.

Una de las aplicaciones más conocidas de las funciones hash es la de cifrar las contraseñas: en un sistema, en vez de almacenar la contraseña *clave*, se guarda su valor hash $h(clave)$. Así, cuando un usuario ingresa su contraseña, el sistema calcula su valor hash y lo compara con el que se tiene guardado. Realizar esto ayuda a evitar que las contraseñas sean conocidas para los usuarios con privilegios, como pueden ser los administradores.

2.4.1. Integridad de datos

Las funciones criptográficas hash también son conocidas como funciones *procesadoras de mensajes* y el valor hash $h(m)$ de un mensaje m dado es llamado *huella* de m; ya que es una representación compacta de m y, dada la resistencia a la segunda preimagen, la huella es prácticamente única. Si el mensaje fuese modificado, el valor hash sería distinto; por lo que si se tienen almacenados los valores hash, basta con calcular su valor $h(m)$ y compararlo con el que se tiene guardado para detectar modificaciones. Por esta razón, las funciones hash también son llamadas códigos de detección de modificaciones (como el Modification Detection Code-2 (MDC-2)).

2.4.2. Firmas

Sea (n, e) la llave pública RSA y d el exponente decodificador secreto de Alice. En el esquema básico de firma RSA, Alice puede firmar mensajes que estén codificados por números $m \in \{0, \ldots, n-1\}$. Para firmar m, aplica el algoritmo de descifrado y obtiene la firma $\sigma = m^d \ mod \ n$ de m. Normalmente, n es un número de 1024 bits y Alice puede firmar una cadena de bits m tal que, cuando es interpretada como número, sea menor que n. Esto es una cadena de, máximo, 128 caracteres American Standard Code for Information Interchange (ASCII): la mayoría de los documentos que se desean firmar suelen ser mucho más grandes. Este problema existe en todos los esquemas de firma digital y usualmente es resuelto al aplicar una función hash resistente a colisiones h. De esta forma, primero se obtiene el valor hash del

mensaje $h(m)$ y esto es lo que se firma en lugar del mensaje mismo (m):

$$\sigma = h(m)^d \quad mod \quad n \tag{2.16}$$

Los mensajes que tengan el mismo valor hash tienen la misma firma. En este caso, es primordial que la función hash h sea resistente a colisiones para garantizar el no repudio. De otra manera, Alice podría firmar el mensaje m y después decir que había firmado un mensaje distinto (n). La resistencia a segundas preimágenes previene que un atacante Eve tome un mensaje m firmado por Alice, genere un mensaje nuevo n y utilice σ como una firma válida de Alice para n.

2.4.3. Message Digest-4 (MD4)

En la década de 1990 esta función hash fue diseñada por Ronald Rivest. Tiene entradas de longitud arbitraria y la longitud de la salida procesada es de 128 bits. El Message Digest-4 (MD4) fue innovador y clave en el diseño para los algoritmos venideros de esta clase (como el MD5).

2.4.4. RIPEMD

Esta función hash, publicada en 1996, está basada en MD4 y fue diseñada por Hans Dobbertin y otros. Consiste en dos formas equivalentes de la función de compresión de MD4. El algoritmo original (RIPEMD-160) devuelve bloques *procesados* de 160 bits; cuando en 1996 Hans descubrió una colisión en dos rondas, se desarrollaron nuevas versiones mejoradas: RIPEMD-128, RIPE-256, RIPE-320; las cuales dan bloques procesados de 128, 256 y 320 bits respectivamente.

2.4.5. *Secure Hash Algorithm (SHA)*

El algoritmo SHA fue publicado por NIST y National Security Agency (NSA) en 1993; este algoritmo produce bloques de 160 bits y fue desarrollado para reemplazar al MD4; sin embargo, poco después de haber sido publicado tuvo que ser quitado por problemas de seguridad. Actualmente, SHA es conocido como SHA-0.

En 1995, SHA-0 fue reemplazado por SHA-1; tiene una salida de la misma longitud que su predecesor y es una de las funciones hash más populares. Hay que destacar que la seguridad que brinda esta función es limitada, pues tiene el mismo nivel que un cifrado por bloques de 80 bits.

En 2002 NIST publicó tres funciones hash más: SHA-256, SHA-384 y SHA-512; esta familia de funciones hash es conocida como SHA-2 y fue desarrollada para cubrir la necesidad de una llave más grande para poder empatar su tamaño con AES. Dos años más tarde, una nueva función hash fue agregada a la familia SHA-2: SHA-224.

Finalmente, en 2008, NIST inició un concurso para buscar al SHA-3 y en 2012 anunció al ganador: Keccak, una función hash desarrollada por Guido Bertoni, Joan Daemen, Michael Peeters y Gilles Van Assche. Esta función tiene una construcción completamente distinta a las familias anteriores.

2.5. Códigos de Autenticación de Mensaje (MAC)

La información presentada a continuación puede consultarse con más profundidad en las siguientes referencias [1], [2], [19].

Las funciones MAC son la técnica simétrica estándar utilizada tanto para la autenticación como para la protección de la integridad de los mensajes. Dependen de unas llaves secretas que son compartidas entre las partes que se van a comunicar; cada una de las partes puede producir el MAC correspondiente para un mensaje dado. Como se explica a continuación, los MAC pueden ser obtenidos mediante cifradores de bloque, cifradores de flujo o de funciones hash criptográficas.

Las funciones hash con llave cuyo propósito específico es la AUTENTICACIÓN DE ORIGEN y garantizar la INTEGRIDAD DE DATOS del mensaje son llamadas MAC. Estas funciones tienen como entrada una llave secreta k y un mensaje de longitud arbitraria y dan como resultado un mensaje de longitud n.

$$h_k : \{0,1\}^* \longrightarrow \{0,1\}^n \tag{2.17}$$

El algoritmo MAC más usado basado en un cifrador de bloque utiliza el modo de operación CBC (véase sección 2.2.10.2). Cuando DES es utilizado como el cifrado de bloque E, el tamaño de bloque es de 64 bits y la llave MAC es de 56 bits.

Otra manera de construir MAC es mediante un algoritmo de MESSAGE DIGEST CIPHER (MDC) que incluya una llave secreta k como parte de la entrada. Un ejemplo de esto es el algoritmo MD5-MAC; donde la función de compresión depende de la llave secreta k, que interviene en todas las iteraciones.

El algoritmo MESSAGE AUTHENTICATOR ALGORITHM (MAA) fue diseñado en 1938 específicamente para obtener MAC en máquinas de 32 bits. El tiempo de ejecución es directamente proporcional a la longitud del mensaje y alrededor de cuatro veces más largo que el MD4.

Pseudorandom Function Es posible generar códigos de verificación mediante el uso de funciones pseudoaleatorias PSEUDORANDOM FUNCTION (PRF). Se define en 2.18 el algoritmo para obtener el código de verificación y en 2.19 el algoritmo para verificar si es correcto o no el MAC.

CBC-MAC Este algoritmo está basado en el modo de operación CBC y una función F que puede ser, por ejemplo, un cifrador por bloques. Se encarga de cifrar con una llave l_1 todo el mensaje m, pero lo único que se toma en cuenta es el último bloque, que es tomado como el código de autenticación (véase figura 2.16). En algunos casos, se cifra de nuevo, con la misma función F, pero utilizando una llave l_2 distinta (véase figura 2.17).

```
entrada:    llave k y mensaje m
salida:     código de autenticación t.
Sea F una función pseudoaleatoria tal que F:(K × X) → Y.
inicio
  t := F(k, m)
  regresar t
fin
```

Pseudocódigo 2.18: MAC mediante PRF, obtener código.

```
entrada:    llave k, mensaje m, código t.
salida:     resultado de la verificación r.
inicio
  si t = F(k, m) entonces:
    regresar verdadero
  si_no:
    regresar falso
fin
```

Pseudocódigo 2.19: MAC mediante PRF, verificar código.

CBC-MAC es considerado seguro cuando el mensaje m tiene una longitud múltiplo del tamaño de bloque del cifrador por bloques. Como se le han encontrado varias vulnerabilidades, algoritmos basados en CBC-MAC han sido propuestos, tales como el *eXtended CBC* o el One-key MAC (OMAC).

Nested MAC Nested MAC (NMAC) es muy parecido al CBC-MAC, sin embargo, este utiliza una función F cuya salida se utiliza como la llave del siguiente bloque. Cuando se termina de procesar el mensaje, se le pone un relleno para completar el bloque y se cifra una vez más, utilizando una llave l_2 (véase figura 2.18). Es importante cifrar el último bloque usando la llave l_2, ya que, de lo contrario, un adversario podría agregar al final nuevos bloques a un mensaje interceptado, calculando un nuevo código de autenticación, dándole como entrada a la función F el código de autenticación original.

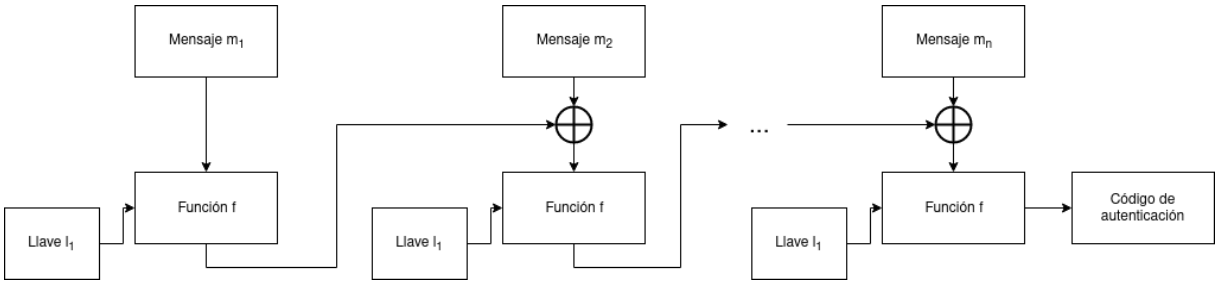


Figura 2.16: Esquema de CBC-MAC simple.

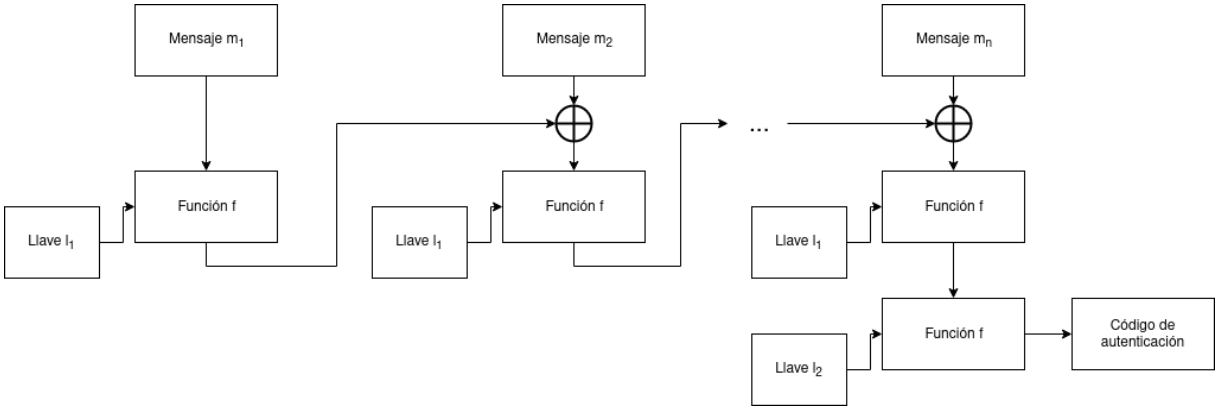


Figura 2.17: Esquema de CBC-MAC con el último bloque cifrado.

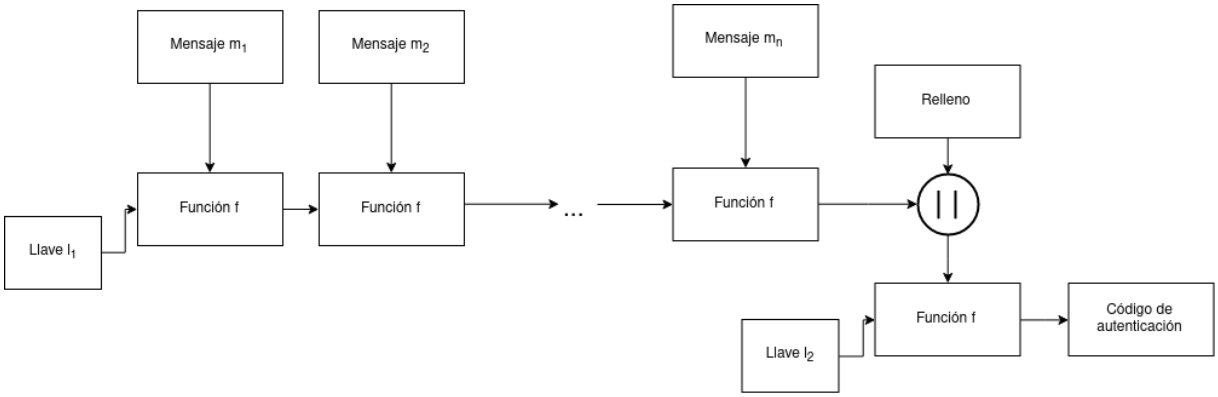


Figura 2.18: Esquema de NMAC.

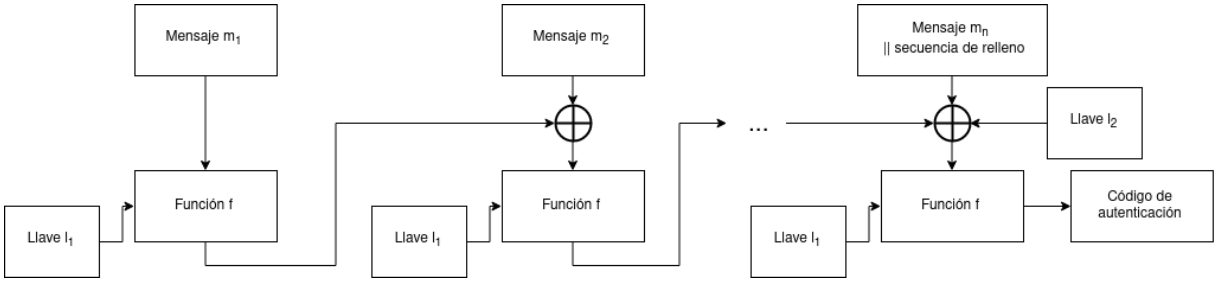


Figura 2.19: Esquema de CMAC.

Cipher-based MAC Parecido al algoritmo CBC-MAC, CIPHER-BASED MAC (CMAC) utiliza una función F y dos llaves (l_1 y l_2); sin embargo, en vez de utilizar l_2 para el segundo cifrado, se concatena con el último bloque del mensaje (véase figura 2.19); entre estos dos elementos (m_n y l_2), se agrega una secuencia de relleno para poder diferenciarlos. Este paso impide que nuevos bloques sean agregados.

El algoritmo OMAC1 es equivalente al CMAC y, en 2005, se convirtió en una recomendación por el NIST.

Parallel MAC PARALLEL MAC (PMAC) utiliza dos llaves (l_1 y l_2) y dos funciones (F_p y F_f); además, a diferencia de los algoritmos descritos anteriormente, PMAC, tal como lo indica su nombre, puede ser ejecutado paralelamente mediante el uso de hilos, pues para procesar un bloque b_x no es necesario b_{x-1}.

La llave l_1 y un contador c_i se cifran utilizando la función F_p, la salida es agregada al mensaje m_i con una XOR; después, se cifra esta salida mediante la función F_f y la llave l_2. Cuando se han procesado todos los bloques, todas las salidas de F_f se combinan mediante una operación XOR y el resultado es cifrado nuevamente mediante F_f, que hace uso de l_2 (véase figura 2.20)

Una de las ventajas de PMAC, es que permite actualizar un bloque b_x fácilmente, sin tener que recalcular la salida de F_P para todos los demás. Es importante que la función F_p sea mucho más rápida que la función F_f.

One-key MAC Este algoritmo, similar a LIBRETA DE UN SOLO USO, es, generalmente, más rápido que los algoritmos basados en funciones PRF. En 2.20 se define el algoritmo para obtener el código de autenticación. La verificación del código se realiza similar a como se mostró en el algoritmo 2.19.

Keyed-Hashed Message Authentication Code Similar a NMAC, este algoritmo utiliza una función h OWHF para producir códigos de autenticación (véase figura 2.21). Tiene tres parámetros: $relleno_e$, $relleno_s$ y una llave l; los primeros dos se encargan de modificar la llave secreta l, por lo que se recomienda que los valores que asumen cambien lo más posible a l mediante la función h.

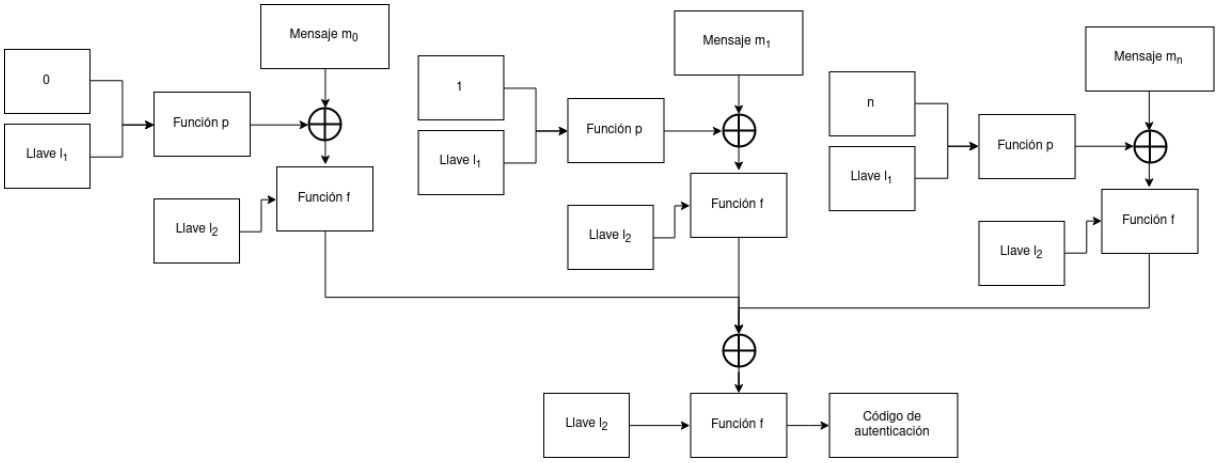


Figura 2.20: Esquema de PMAC.

```
1  entrada:    llaves l_1, l_2 que se encuentran en el intervalo [1,q]
2              mensaje M = m_L m_L - 1 ... m_1
3  salida:     código de autenticación t.
4  Sea P una función tal que P(m,x) = m_L x^L + ... + m_1 x.
5  Sea q un primo de aproximadamente 2^128.
6  inicio
7    t := (P(m,l_1) + l_2)  mód q
8    regresar t
9  fin
```

Pseudocódigo 2.20: MAC mediante *One-time* MAC.

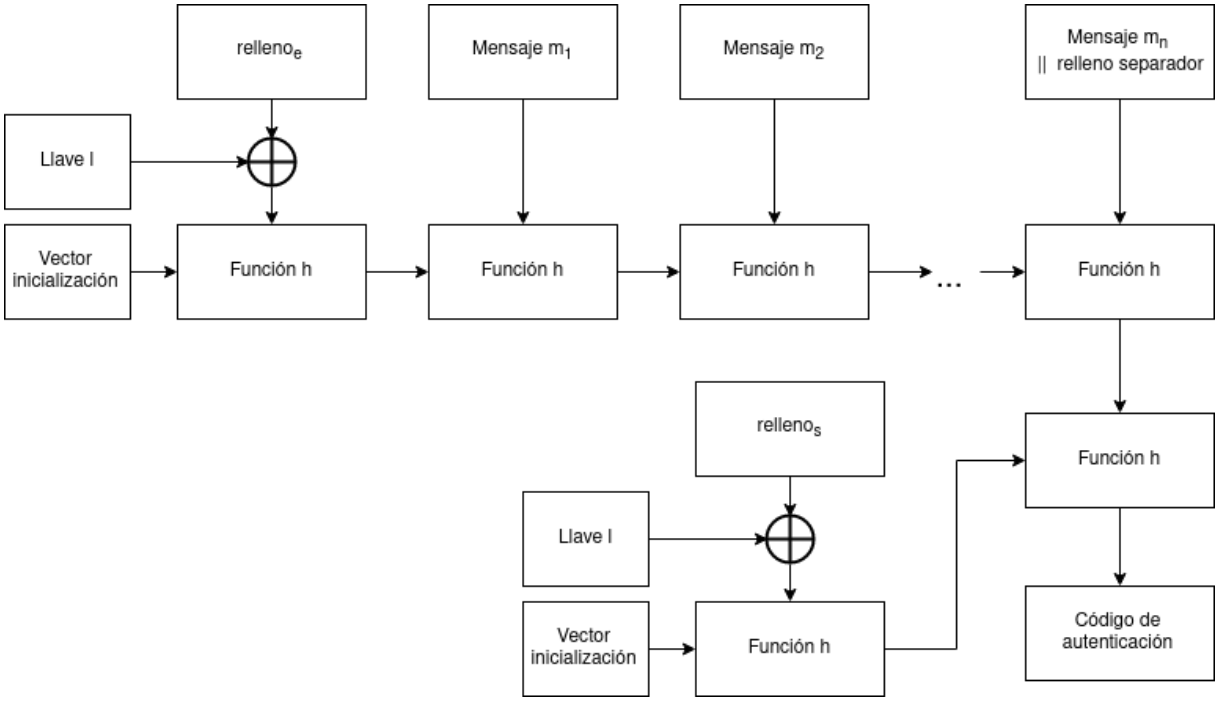


Figura 2.21: Esquema de HMAC.

Este algoritmo es ampliamente utilizado, por ejemplo, se encuentra presente en los protocolos SSL y Secure Shell (SSH).

2.6. *Tweakable Encyphering Eschemes* (TES)

La información que aquí se presenta puede ser consultada con mayor detalle en [20] y [21].

La principal motivación para el diseño de Tweakable Encyphering Scheme (TES) son los Tweakable Block Cipher (TBC), los cuales fueron pensados como un medio de proveer de variabilidad a los cifrados por bloques. En el esquema original, un cifrado por bloques es totalmente determinístico: para el mismo mensaje y la misma llave, el texto cifrado generado es siempre el mismo. Los TBC agregan un *tweak* a la entrada de un cifrador por bloques para darle variabilidad; de esta manera, una misma llave y un mismo mensaje, ya no producen siempre el mismo texto cifrado. El papel del *tweak* es bastante similar al del vector de inicialización, solo que este último opera a nivel de los modos de operación.

Es importante resaltar las diferencias entre el papel del *tweak* y el papel de la llave: esta provee de incertidumbre a un adversario, mientras que el *tweak* proporciona variabilidad. Mantener el *tweak* en secreto no debería proporcionar mayores niveles de seguridad; de hecho, una condición en el diseño de TBC es que la seguridad del cifrado por bloques no se ve incrementada (ni decrementada) por la introducción del *tweak*.

En la ecuación 2.18 se muestra la firma para un cifrado por bloques con un *tweak*. Comparando esta función con la de la ecuación 2.3 (véase sección de cifrados por bloques, 2.2) se agrega un conjunto más al producto cruz de la entrada: una cadena de bits de tamaño t (el espacio de los *tweaks*).

$$\tilde{E} : \{0,1\}^k \times \{0,1\}^y \times \{0,1\}^n \longrightarrow \{0,1\}^n \tag{2.18}$$

En las siguientes ecuaciones se muestran algunas de las construcciones para TBC propuestas a la fecha:

$$\tilde{E}_k(T, M) = E_k(T \oplus E_k(M)) \tag{2.19}$$

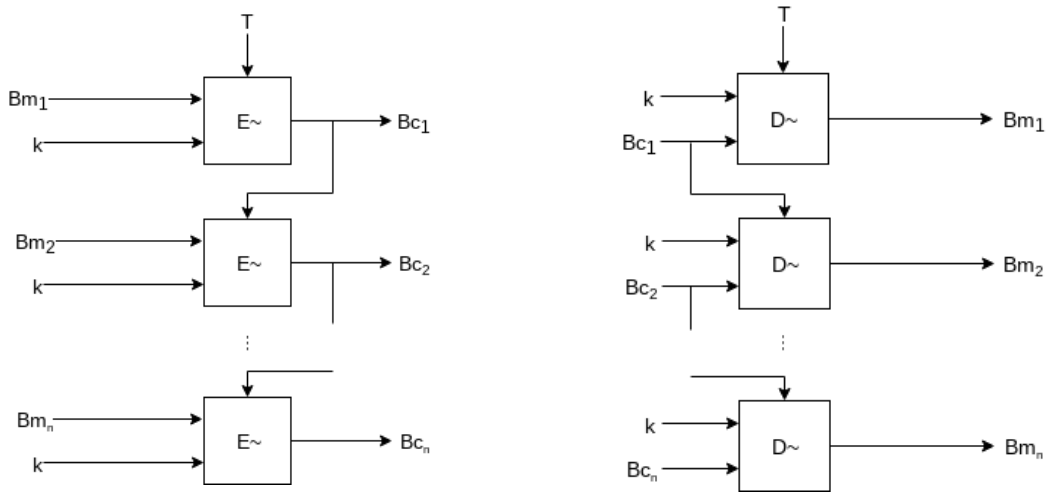
$$\tilde{E}_{k,h}(T, M) = E_k(M \oplus h(T)) \oplus h(T) \tag{2.20}$$

$$\tilde{E}_k(T, M) = E_k(M \oplus E_k(T)) \tag{2.21}$$

$$\tilde{E}_k(T, M) = E_k(M \oplus E_k(T)) \oplus E_k(T) \tag{2.22}$$

Con los TBC se pueden crear modos de operación análogos a los que se usan con los cifrados por bloques estándares. En la figura 2.22 el Tweakable Block Chaining (TBC), que es análogo a CBC (sección 2.2.10.2).

Los TBC son usados para la construcción de TES. Para estos existen dos clasificaciones: *encrypt-mask-encrypt* y *hash-counter-hash*. En la primera clasificación se encuentran aquellos que cuentan con dos capas de cifrado y una de enmascaramiento; algunos ejemplos de esta son CBC-Mask-CBC (CMC), ECB-Mask-ECB (EME) y Arbitrary Block Length Mode (ABL). La segunda consiste en dos



(a) Cifrado.

(b) Descifrado.

Figura 2.22: Modo de operación TBC.

funciones hash con un cifrado por bloques en modo de operación de contador (sección 2.2.10.5); algunos ejemplos son Extended Codebook (XCB), Hash CTR (HCTR) y HCH.

La cuestión que queda por esclarecer ahora es, ¿cuál es la ventaja de utilizar TBC en lugar de cifrados por bloques normales?, o en otro nivel, ¿cuál es la ventaja de utilizar modos de operación basados en *tweaks* a modos de operación con vectores de inicialización? Una primera impresión es que se trata de lo mismo: ambos, el *tweak* y el vector de inicialización, son entradas extras que permiten introducir variabilidad en los cifrados por bloques.

El primer punto clave es que los *tweaks* introducen una división semántica más en el análisis y diseño de modos de operación. En el esquema tradicional solo existen dos divisiónes: el nivel bajo, referente a los cifrados por bloques; y el nivel alto, referente a los modos de operación. Con los *tweaks*, el problema de los modos de operación se divide en dos: cómo diseñar buenos TBC, y cómo diseñar buenos modos de operación basados en TBC. Este nuevo enfoque permite filtrar nuevos problemas en un nivel en particular sin que se afecten a las construcciones de los otros niveles. El segundo punto clave es que los TES pueden ser diseñados para recibir mensajesde longitud arbitraria y regresar cifrados de su misma longitud.

2.7. Cifrados que preservan el formato

El objetivo de los cifrados que preservan el formato (en inglés, Format Preserving Encryption (FPE)) está bastante bien definido: lograr que los mensajes cifrados *se vean* igual que los mensajes en claro. Obviamente para que esto tenga sentido hay que definir qué es lo que hace que un mensaje se parezca a otro; los cifradores tradicionales, como AES, reciben cadenas binarias y entregan cadenas binarias, por lo que, en cierto sentido, son ya un tipo de cifrados que preservan el formato. En una clase más general de algoritmos FPE el formato debe ser una parámetro: cadenas binarias, caracteres ASCII imprimibles, dígitos decimales, dígitos hexadecimales, etcétera.

Desde un inicio, los cifrados que preservan el formato se perfilaron como una posible solución para el problema de la tokenización: usando un alfabeto de dígitos decimales se logra que los *tokens* y los Personal Account Number (PAN) tengan el mismo aspecto. De las posibles soluciones presentadas en la sección 3.1.4, FPE es la que presenta un esquema más tradicional dentro de la criptografía, ya que el proceso de cifrado y descifrado es el mismo que un cifrador simétrico.

La utilidad de los cifrados que preservan el formato se centra principalmente en *agregar* seguridad a sistemas y protocolos que ya se encuentran en un entorno de producción. Estos son algunos ejemplos de dominios comunes en FPE:

- Números de tarjetas de crédito.
  `5827 5423 6584 2154` → `6512 8417 6398 7423`
- Números de teléfono.
  `55 55 54 75 65` → `55 55 12 36 98`
- CURP.
  `GHUJ887565HGBTOK01` → `QRGH874528JUHY01`

2.7.1. Clasificación de los cifrados que preservan el formato

En [6] Phillip Rogaway propone una clasificación para los cifrados que preservan el formato que se basa en el tamaño del espacio de mensajes ($N = |X|$):

Espacios minúsculos El espacio es tan pequeño que es aceptable gastar $O(N)$ en un preproceso de cifrado. Esto es, cifrar todos los posibles mensajes de una sola vez, para que las subsiguientes solicitudes de cifrado y descifrado consistan en simples consultas a una base de datos.

El tamaño de N depende del contexto en el que se vaya a utilizar el algoritmo; para el contexto del problema de la tokenización ($N \approx 10^{16}$) no resulta viable utilizar esta técnica.

```
1  entrada: lista l[0,...,n-1]
2  salida: misma lista barajeada
3  inicio
4    para_todo i desde n-1 hasta 0:
5      j ← rand(0, i)
6      swap(l[j], l[i])
7    fin
8  fin
```

Pseudocódigo 2.21: *Knuth shuffle*, [22].

Algunos ejemplos de cómo hacer el preproceso de cifrado son el *Knuth shuffle* (también conocido como *Fisher-Yates shuffle*, pseudocódigo 2.21) o un CIFRADO CON PREFIJO.

Espacios pequeños En esta clasificación se colocan a los espacios de mensaje cuyo tamaño no es más grande que 2^w en donde w es el tamaño de bloque del cifrado subyacente. Para AES, en donde $w = 128$, $N = 2^{128} \approx 10^{38}$.

En este esquema, el mensaje se ve como una cadena de n elementos pertenecientes a un alfabeto de cardinalidad m (i. e. $N = m^n$). Por ejemplo, para números de tarjetas de crédito, $n \approx 16$ y $m = 10$, por lo que $N = 10^{16}$ (diez mil trillones); lo cual es aproximadamente 2.93×10^{-21} % de 2^{128}.

Los algoritmos que preservan el formato expuestos en la sección 3.1.4 pertenencen a esta categoría.

Espacios grandes El espacio es más grande que 2^w. Para estos casos, el mensaje se ve como una cadena binaria. Las técnicas utilizadas incluyen cualquier cifrado cuya salida sea de la misma longitud que la entrada (p. ej. los TES: CMC, EME, HCH, etc. Sección 2.6).

Como se puede observar de los ejemplos dados, el problema de la tokenización de números de tarjetas de crédito es un problema de espacios pequeños.

Figura 2.23: Componentes de un número de tarjeta.

2.8. Composición del número de una tarjeta

La información presentada a continuación puede consultarse con más detalle en los siguientes documentos [23]-[25].

Como se puede observar en la figura 2.23, un número de tarjeta PAN, se compone por tres partes: el número identificador del emisor (Issuer Identification Number (INN)), el número de cuenta y un dígito verificador; la longitud del PAN puede variar e ir desde los 12 hasta los 19 dígitos. A continuación se explica con más detalle cada uno de sus componentes

2.8.1. Identificador del emisor

El INN comprende los primeros 6 dígitos del número de tarjeta. El primer dígito es conocido como Major Industry Identifier (MII) y se encarga de identificar la industria a la que pertenece el emisor; en la tabla 2.2, se puede observar la relación entre el dígito y el giro. El identificador del emisor provee, entre otros, los siguientes datos:

1. Emisor de la tarjeta

2. Tipo de la tarjeta (ej. crédito o débito)

3. Nivel de la tarjeta (ej. clásica, Gold, Black)

2.8.2. Número de cuenta

Todos los dígitos posteriores al INN y anteriores al último dígito, comprenden el número de cuenta. La longitud de este puede variar, pero, máximo comprende 12 dígitos, por lo que cada emisor tiene 10^{12} posibles números de cuenta.

Dígito	Industria
1, 2	Aerolíneas
3	Viajes y entretenimiento (ej. American Express)
4, 5	Bancos e industria financiera (ej. Visa, Mastercard)
6	Comercio (ej. Discover)
7	Industria petrolera
8	Telecomunicaciones
9	Asignación nacional

Tabla 2.2: Identificador de industria (MII).

2.8.3. Dígito verificador

Finalmente, se tiene el dígito verificador; este toma en cuenta todos los dígitos anteriores y se calcula mediante el algoritmo de Luhn, que se describe en 2.22.

```
1  entrada: número de tarjeta, compuesto desde 12 hasta 19 dígitos:
2           {x_n x_{n-1} x_{n-2} ... x_3 x_2 x_1}
3  salida: dígito verificador x_1.
4  inicio
5    Sean los conjuntos x_par y x_impar:
6        si n es par:
7            x_par = {x_2, x_4, x_6, ..., x_n}
8            x_impar = {x_3, x_5, x_7, ..., x_{n-1}}.
9        si_no:
10           x_par = {x_2, x_4, x_6, ..., x_{n-1}}
11           x_impar = {x_3, x_5, x_7, ..., x_n}.
12   Obtener el doble de cada uno de los elementos del conjunto x_par:
13     x_pardoble = {2 × x_2, 2 × x_4, 2 × x_6, ...}.
14   ∀x_i ∈ x_pardoble > 9:
15       x_i = (x_i mód 10) + 1
16   Sumar los elementos de los conjuntos x_pardoble y x_impar.
17   x_1 = (S × 9) mód 10
18   Regresar x_1
19 fin
```

Pseudocódigo 2.22: Algoritmo de Luhn.

Capítulo 3

Análisis y diseño

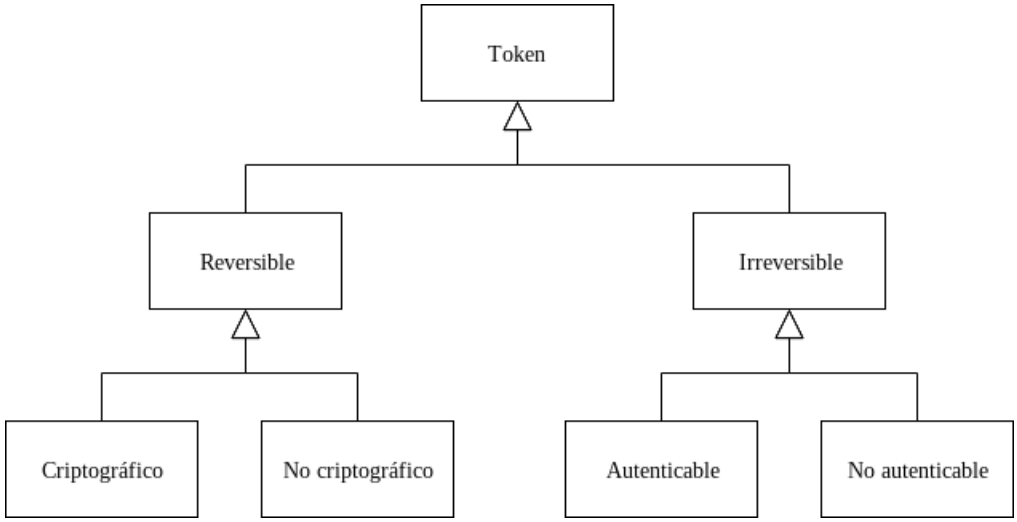


Figura 3.1: Clasificación de los TOKENS.

3.1. Generación de *tokens*

3.1.1. Requerimientos

En [26], el PAYMENT CARD INDUSTRY (PCI) SECURITY STANDARD COUNCIL (SSC) divide a los TOKENS en reversibles e irreversibles. A su vez, los reversibles se dividen en criptográficos y en no criptográficos; mientras que los irreversibles se dividen en autenticables y no autenticables (figura 3.1).

Los TOKENS irreversibles no pueden, bajo ninguna circunstancia, ser reconvertidos al PAN original. Esta restricción aplica tanto para cualquier entidad en el entorno del negocio (comerciante, proveedor de TOKENS, banco) como para cualquier posible atacante. Dados un PAN y un TOKEN, los identificables permiten validar cuando el primero fue utilizado para la creación del segundo, mientras que los no identificables, no.

La clasificación del PCI SSC con respecto a los reversibles resulta un poco confusa (esto ya ha sido señalado antes, [27]). Establece que los criptográficos son generados utilizando CRIPTOGRAFÍA FUERTE, el PAN nunca se almacena, solamente se guarda una llave; los no criptográficos guardan la relación entre TOKENS y PAN en una base de datos. El problema está en que no se menciona *cómo* generar los no criptográficos. A pesar del nombre, los métodos más comunes para esta categoría ocupan PRIMITIVAS CRIPTOGRÁFICAS (p. ej. generadores pseudoaleatorios); además de que, en una implementación real, para poder cumplir con el PCI DATA SECURITY STANDARD (DSS), la propia base de datos debe de estar cifrada [28].

PCI DSS define cuatro *dominios* de seguridad para el proceso de tokenización:

1. **Generación de tokens.** Para cada clase tokenizadora, este dominio se encarga de definir consideraciones para generación segura de TOKENS. Cubre los dispositivos, procesos, mecanismos y algoritmos que son utilizados para crear los TOKENS.

2. **Mapeo de tokens.** Este dominio, que se refiere al mapeo de los TOKENS con su PAN origen, aplica solamente a los procesos de tokenización reversibles. Entre otras cosas, provee guías respecto a control de acceso necesarios para las peticiones de tokenización.

3. **Bóveda de datos de tarjeta.** Como el dominio pasado, solo aplica a las implementaciones de tokenización reversibles. Cubre el cifrado obligado del PAN y los controles de acceso necesarios para entrar a la CARD DATA VAULT (CDV).

4. **Manejo criptográfico de llaves.** Define las buenas prácticas para el manejo criptográfico de las llaves y las operaciones realizadas con ellas por el producto tokenizador.

En esta sección se explica cada una de las categorías y se analizan los requerimientos que deben tener. Para comenzar, se enlistan los requerimientos aplicables a todos los TOKENS, sin importar su categoría:

REQPCI-01 Validación de productos de hardware.
Si se usa un producto de hardware para la tokenización, este debe de ser validado por FIPS 140-2 nivel 3 o superior (descrito en [29]).

REQPCI-02 Validación de productos de software.
Si se usa un producto de software para la tokenización, este debe de ser validado por FIPS 140-2 nivel 2 o superior (descrito en [29]).

REQPCI-03 Resistencia a texto claro conocido.
Un atacante con acceso a múltiples pares de TOKENS y PAN no debe de ser capaz de determinar otros PAN a partir de solamente TOKENS. En otras palabras, los TOKENS deben ser resistentes a ataques con texto en claro conocido (sección 2.1.2).

REQPCI-04 Resistencia a sólo texto cifrado.
Recuperar un PAN a partir de un TOKEN debe de ser COMPUTACIONALMENTE NO FACTIBLE (resistencia a ataques con sólo texto cifrado, sección 2.1.2).

REQPCI-05 Detección de anomalías.
Se deben de implementar disparadores que permitan detectar irregularidades en el sistema (anomalías, funcionamientos erróneos, comportamientos sospechosos). El producto debe registrar dichos eventos y avisar al personal correspondiente.

REQPCI-06 Distinción entre tokens y PAN.
Se debe contar con un mecanismo para distinguir entre TOKENS y PAN. Los proveedores del servicio

de tokenización deben compartir este mecanismo con la entidad (o entidades) que usa los TOKENS.

REQPCI-07 Guía de instalación.

Se debe de contar con una guía de instalación y uso para el correcto funcionamiento del producto de tokenización.

REQPCI-08 Integridad del proceso de tokenización.

Deben de implementarse mecanismos que garanticen la integridad del proceso de generación de TOKENS.

REQPCI-09 Acceso al proceso de tokenización.

Solo los usuarios autenticados y componentes del sistema tienen permitido acceder al proceso de tokenización y de-tokenización. Los métodos utilizados deben ser al menos tan rigurosos como lo indicado en el requerimiento 8 del PCI DSS [28].

SUBREQPCI-09/1 Control de peticiones.

Todas las peticiones deben pasar a través de una APPLICATION PROGRAM INTERFACE (API) que controle todos los intentos de acceso y aplique de manera uniforma reglas de control de acceso.

SUBREQPCI-09/2 Registros de acceso.

Se deben registrar todos los eventos de acceso, de tokenización y de detokenización. Esta funcionalidad debe ser configurable de manera segura. Para esto se debe seguir el requerimiento 4 del PAYMENT APPLICATION (PA)-DSS [30].

SUBREQPCI-09/3 Autenticación multifactor.

El sistema debe soportar AUTENTICACIÓN MULTIFACTOR para todos los tipos de usuario: accesos administrativos, operaciones de tokenización y detokenización, mantenimiento, etcétera.

SUBREQPCI-09/4 Accesos a nivel de sistema.

Todos los accesos a nivel de sistema deben soportar AUTENTICACIÓN MUTUA, incluyendo a las peticiones de tokenización y detokenización.

SUBREQPCI-09/5 Accesos administrativos.

Se debe utilizar CRIPTOGRAFÍA FUERTE para todos los accesos administrativos que no se hagan desde consola.

REQPCI-10 Mapeos de token a token prohibidos.

No se debe poder pasar de un primer TOKEN válido a un segundo, también válido; forzosamente debe existir un estado intermedio: del primer TOKEN se pasa al PAN correspondiente (operación de detokenización) y de este se pasa al segundo TOKEN.

REQPCI-11 Protección contra vulnerabilidades comunes.

Se deben implementar medidas en contra de las vulnerabilidades de seguridad más comunes ([30], requerimiento 5.2). Algunas de estas medidas pueden ser el uso de herramientas de análisis de código estático, o el uso de lenguajes de programación especializados.

REQPCI-12 Primitivas criptográficas usadas.

Las primitivas criptográficas que se usen deben estar basadas en estándares nacionales (referentes a Estados Unidos) o internacionales (p. ej. AES). Ver sección 3.1.1.4.

REQPCI-13 Sobre el manejo adecuado de llaves.

En donde se usen llaves para la generación y protección de TOKEN, se deben seguir buenas prácticas criptográficas para la administración de estas. En particular, se deben cumplir con las recomendaciones del NIST en [31] y [32].

SUBREQPCI-13/1 Sobre el ciclo de vida.

La llave tokenizadora debe seguir la política de los ciclos de llaves descritos en el ISO/IEC 115681 (ver sección 3.1.2.1).

SUBREQPCI-13/2 Descripción del periodo criptográfico activo.

La política sobre el tiempo de vida de la llave debe incluir una descripción sobre el periodo criptográfico activo de la llave tokenizadora en cuestión.

SUBREQPCI-13/3 Sobre la destrucción de las llaves.

El proveedor debe incorporar una función que permita la destrucción de sus llaves criptográficas sin tener que alterar o abrir el dispositivo.

SUBREQPCI-13/4 Exportar llave en claro prohibido.

Las llaves usadas para generar TOKENS no se deben poder exportar en claro desde el programa.

SUBREQPCI-13/5 Entropía de generación de llaves.

La fuente generadora de llaves debe tener, al menos, 128 bits de ENTROPÍA.

SUBREQPCI-13/6 Llaves de uso único.

Las llaves criptográficas usadas para generar TOKENS no deben ser usadas para ningún otro fin.

3.1.1.1. Irreversibles

REQPCI-14 Sobre la generación de tokens (irreversibles).

El mecanismo utilizado para la generación de los TOKENS no es reversible (o improbable).

SUBREQPCI-14/1 Sobre el mecanismo generador (irreversibles).

El proceso para crear TOKENS clasificado como irreversible debe asegurar que el mecanismo, proceso o algoritmo utilizado para crear el TOKEN no sea reversible. Si una función hash (véase sección 2.4) es utilizada, esta debe ser una PRIMITIVA CRIPTOGRÁFICA y utilizar una llave secreta k tal que el mero conocimiento de la función hash no permita la creación de un ORÁCULO.

SUBREQPCI-14/2 Contenido en claro (irreversibles).

Los TOKENS irreversibles no deben contener dígitos en claro del PAN original, excepto que estos dígitos sean una coincidencia.

SUBREQPCI-14/3 Creación de un diccionario (irreversibles).

La creación de una tabla o *diccionario* de TOKENS estáticos debería ser imposible, o, al menos, al punto de satisfacer que la probabilidad de predecir correctamente el PAN sea menor que $\frac{1}{10^6}$.

SUBREQPCI-14/4 Sobre el proceso de autenticación (irreversibles).

En el caso de los TOKENS autenticables, el proceso de autenticación no debe revelar información suficiente para realizar búsquedas, excepto una exhaustiva (PAN por PAN) y se deben implementar controles para detectar estas últimas.

3.1.1.2. Criptográficos reversibles

REQPCI-15 Probabilidad de adivinar relaciones (criptográficos).

La probabilidad de adivinar la relación entre un TOKEN y un PAN debe de ser menor que 1 en 10^6.

SUBREQPCI-15/1 Distribución uniforme (criptográficos).

Para un PAN dado, todos los TOKENS deben ser equiprobables; esto es, el mecanismo tokenizador no debe exhibir tendencias probabilísticas que lo expongan a ataques estadísticos.

SUBREQPCI-15/2 Permutación aleatoria (criptográficos).

El método de tokenización debe actuar como una familia de PERMUTACIONES aleatoria desde el espacio de PAN al espacio de TOKENS.

SUBREQPCI-15/3 Cambio de llave (criptográficos).

Un cambio en la llave se debe ver reflejado en un cambio en el TOKEN resultado.

SUBREQPCI-15/4 Cambio de PAN (criptográficos).

Un cambio en el PAN se debe ver reflejado en un cambio en el TOKEN resultado.

SUBREQPCI-15/5 Verificación de la aleatoriedad (criptográficos).

Se debe tener un medio para verificar de forma práctica la aleatorización de dígitos, de acuerdo a lo establecido en NIST 800-90A [33].

REQPCI-16 Almacenamiento de tokens (criptográficos).

Los TOKENS generados no se deben almacenar en ningún punto del sistema.

El requerimiento anterior (REQPCI-16 ALMACENAMIENTO DE TOKENS (CRIPTOGRÁFICOS), RC1C en [26]) es un tanto difícil de interpretar; la versión original establece: *los* TOKENS *basados en el* PAN *completo no se deben almacenar si el producto tokenizador también almacena su* PAN *truncado correspondiente*. Es un requerimiento de los criptográficos reversibles, por lo que el TOKEN no se debería almacenar bajo ninguna circunstancia (según la propia clasificación del PCI SSC); la redacción del requerimiento se cambió para reflejar este hecho.

REQPCI-17 Seguridad de la administración de llaves (criptográficos).

Todas la operaciones sobre la administración de las llaves criptográficas deben realizarse en un dispositivo criptográfico seguro y aprobado: el PCI SSC se encarga de hacer validaciones; también puede ser cualquier dispositivo validado por FIPS 140-2 nivel 3 o superior [29] o por la INTERNATIONAL ORGANIZATION FOR STANDARIDIZATION (ISO) 13491-1.

REQPCI-18 Sobre la longitud de las llaves (criptográficos).

Las llaves para *tokenizar* deben tener una FUERZA EFECTIVA de, al menos, 128 bits. Cualquier llave utilizada para proteger o para derivar la llave del TOKEN debe de ser de igual o mayor FUERZA EFECTIVA.

REQPCI-19 Independencia estadística (criptográficos).

Si el espacio de llaves es usado para producir TOKENS es dos contextos distintos (p. ej. para distintos comerciantes), estas deben ser ESTADÍSTICAMENTE INDEPENDIENTES.

3.1.1.3. No criptográficos reversibles

REQPCI-20 Generación y almacenamiento de tokens (no criptográficos).

La generación de un TOKEN debe realizarse independientemente de su PAN, y la relación entre un PAN y su TOKEN sólo tiene que estar almacenada en la base de datos (CDV) establecida.

REQPCI-21 Probabilidad de encontrar un PAN (no criptográficos).

La probabilidad de encontrar un PAN a partir de su respectivo TOKEN debe de ser menor que 1 en 10^6.

SUBREQPCI-21/1 Distribución equiprobable (no criptográficos).

Para un PAN dado, todos sus TOKENS respectivos deben ser EQUIPROBABLES, esto es que el siste-

ma *tokenizador* no debe exhibir patrones probabilísticos que lo vulneren a un ataque estadístico.

SUBREQPCI-21/2 Permutaciones aleatorias (no criptográficos).
El método de tokenización debe actuar como una familia de PERMUTACIONES aleatoria en el espacio efectivo de los PANs al espacio de TOKENS.

SUBREQPCI-21/3 Parámetros de tokenización (no criptográficos).
El método de tokenización debe incluir parámetros tales que, un cambio en estos parámetros resulte en un TOKEN diferente; por ejemplo, un cambio en la instancia del proceso debe derivar en una secuencia de TOKENS distintos, incluso cuando es usada la misma secuencia de TOKENS.

SUBREQPCI-21/4 Verificación de la aleatoriedad (no criptográficos).
Se debe tener un medio para verificar de forma práctica la aleatorización de dígitos, de acuerdo a lo establecido en NIST 800-90A [33].

En [26] se establece un subrequerimiento más de REQPCI-21 PROBABILIDAD DE ENCONTRAR UN PAN (NO CRIPTOGRÁFICOS): *Al cambiar parte de un* PAN*, debe cambiar su* TOKEN *resultante*. Es un requerimiento análogo a SUBREQPCI-15/4 CAMBIO DE PAN (CRIPTOGRÁFICOS), sin embargo en el contexto de los no criptográficos reversibles, tal restricción no tiene sentido, dado que la generación del TOKEN es independiente del PAN (requerimiento REQPCI-20 GENERACIÓN Y ALMACENAMIENTO DE TOKENS (NO CRIPTOGRÁFICOS)).

REQPCI-22 Distribución imparcial (no criptográficos).
El proceso de generación de TOKENS debe garantizar una distribución de TOKENS imparcial, esto significa que la probabilidad de cualquier par PAN/TOKEN debe ser igual.

REQPCI-23 Instancias estadísticamente independientes (no criptográficos).
Si varias o diferentes instancias de la bases de datos (CDV) son usadas, cada una de estas debe ser ESTADÍSTICAMENTE INDEPENDIENTES.

REQPCI-24 Proceso de detokenización (no criptográficos).
El proceso de detokenización debe realizarse por medio de una búsqueda de datos o un índice dentro de la base de datos (CDV), y no por medio de métodos criptográficos.

Un subrequerimiento de REQPCI-24 PROCESO DE DETOKENIZACIÓN (NO CRIPTOGRÁFICOS) que aquí se omite establece: «*El* PAN *y el* TOKEN *debe ser probabilísticamente independientes. Cualquier método lógico o matemático no debe ser usado para tokenizar el* PAN *o detokenizar el* TOKEN». La independencia entre PAN y TOKEN ya se establece en REQPCI-20 GENERACIÓN Y ALMACENAMIENTO DE TOKENS (NO CRIPTOGRÁFICOS). No es clara la ascepción de *método lógico matemático*; una solución común para generar TOKENS no criptográficos es usar PSEUDORANDOM NUMBER GENERATOR (PRNG), los cuales

Algoritmo	Tamaño de llave	Modo de operación
AES	128	CTR, OCB, CBC, OFB, CFB
RSA	3072	RSAES-OAEP
ECC	256	ECDH, ECMQV, ECDSA, ECIES
DSA/DH	3072/256	DHE

Tabla 3.1: Longitudes de llave mínimas y MODOS DE OPERACIÓN permitidos para algoritmos criptográficos

son métodos matemáticos.

REQPCI-25 Cifrado de la base de datos (no criptográficos).

Dentro de la base de datos (CDV), los PAN deben ser cifrados con una llave de mínimo 128 bits de FUERZA EFECTIVA.

REQPCI-26 Seguridad de la administración de llaves (no criptográficos).

Todas la operaciones sobre la administración de las llaves criptográficas deben realizarse en un dispositivo criptográfico seguro y aprobado: el PCI SSC se encarga de hacer validaciones; también puede ser cualquier dispositivo validado por FIPS 140-2 nivel 3 o superior [29] o por la ISO 13491-1.

3.1.1.4. Primitivas criptográficas

En esta sección se resumen los requerimientos mínimos que debe de tener cualquier PRIMITIVA CRIPTOGRÁFICA que se use dentro del sistema *tokenizador*. Esta información se presenta en el anexo C de [26], el cual está clasificado como *informativo* solamente (no *normativo*). Con respecto a las PRIMITIVAS CRIPTOGRÁFICAS, la parte *normativa* está controlada por el programa CRYPTOGRAPHIC ALGORITHM VALIDATION PROGRAM (CAVP) del NIST; el cual evalúa las implementaciones usadas de una serie de primitivas comunes[1].

En la tabla 3.1 se colocan los tamaños mínimos de llaves y MODOS DE OPERACIÓN (sección 2.2.10) asociados para las PRIMITIVAS CRIPTOGRÁFICAS de los «algoritmos criptográficos»[2] permitidos. Estos son: AES (sección 2.2.5), RSA (sección 2.1.3), ELLIPTIC CURVE CRYPTOSYSTEM (ECC) y DIGITAL SIGNATURE ALGORITHM (DSA)/DIFFIE-HELLMAN (DH). Se hace especial énfasis en que TRIPLE DES (TDES) no está permitido.

La tabla 3.2 enlista los algoritmos hash (sección 2.4) permitidos. Para evitar introducir fallas de seguridad a través de las funciones hash, estas deben proveer al menos tantos bits de seguridad como el algoritmo criptográfico usado, y en cualquier caso, no menos de 128 bits (lo que deja fuera a MD4 y MD5).

[1] Esta lista se puede encontrar en HTTPS://CSRC.NIST.GOV/PROJECTS/CRYPTOGRAPHIC-ALGORITHM-VALIDATION-PROGRAM

[2] El PCI SSC parece dividir a las PRIMITIVAS CRIPTOGRÁFICAS en *algoritmos criptográficos*, *funciones hash* y *generadores de números pseudoaleatorios*; esto resulta confuso dado que las tres categorías pertenecen al campo de estudio de la criptografía.

Bits de seguridad	**Algoritmo hash**
128	SHA-256
128	SHA3-256
192	SHA3-384
256	SHA-512
256	SHA3-512

Tabla 3.2: Algoritmos hash permitidos

El número de bits de entropía utilizados para los generadores de números aleatorios debe de ser mayor o igual al número de bits de seguridad utilizados para las primitivas anteriores. Cuando se utilicen generadores determinísticos, estos deben seguir las recomendaciones del NIST en [33].

Los siguientes requerimientos son referentes al cumplimiento de distintos estándares y recomendaciones (principalmente del NIST); en la sección 3.1.2 se resume el contenido de cada uno de estos:

- REQPCI-01 Validación de productos de hardware.
- REQPCI-02 Validación de productos de software.
- REQPCI-09 Acceso al proceso de tokenización.
- REQPCI-13 Sobre el manejo adecuado de llaves.
- SUBREQPCI-15/5 Verificación de la aleatoriedad (criptográficos).

La tabla 3.3 es una relación entre la lista de requerimientos aquí presentada y la notación del PCI SSC en [26]. Sobre todo en cuanto a los requerimientos REQPCI-09 Acceso al proceso de tokenización y REQPCI-13 Sobre el manejo adecuado de llaves el documento del PCI es bastante repetitivo: se colocan 3 versiones (una para cada categoría) con prácticamente el mismo contenido.

Requerimiento	**Equivalente PCI**
REQPCI-01 Validación de productos de hardware	GT1
REQPCI-02 Validación de productos de software	GT3
REQPCI-03 Resistencia a texto claro conocido	GT4
REQPCI-04 Resistencia a sólo texto cifrado	GT5
REQPCI-05 Detección de anomalías	GT6
REQPCI-06 Distinción entre tokens y PAN	GT7
REQPCI-07 Guía de instalación	GT8
Continúa en siguiente página	

Continuación	
Requerimiento	**Equivalente PCI**
REQPCI-08 Integridad del proceso de tokenización	GT9
SUBREQPCI-09/1 Control de peticiones	GT10.1, RC2A-1, RC2A-2 RN2B y RN3B
SUBREQPCI-09/2 Registros de acceso	GT10.2
SUBREQPCI-09/3 Autenticación multifactor	GT10.3
SUBREQPCI-09/4 Accesos a nivel de sistema	GT10.4
SUBREQPCI-09/5 Accesos administrativos	GT10.5
REQPCI-10 Mapeos de token a token prohibidos	GT11
REQPCI-11 Protección contra vulnerabilidades comunes	GT12
REQPCI-12 Primitivas criptográficas usadas	GT13
SUBREQPCI-13/1 Sobre el ciclo de vida	IT4A-1 y RC4B-1
SUBREQPCI-13/2 Descripción del periodo criptográfico activo	IT4A-2 y RC4B-2
SUBREQPCI-13/3 Sobre la destrucción de las llaves	IT4A-3 y RC4B-3
SUBREQPCI-13/4 Exportar llave en claro prohibido	RC1A-1
SUBREQPCI-13/5 Entropía de generación de llaves	RC1A-2
SUBREQPCI-13/6 Llaves de uso único	RC1A-3
SUBREQPCI-14/1 Sobre el mecanismo generador (irreversibles)	IT1A-1
SUBREQPCI-14/2 Contenido en claro (irreversibles)	IT1A-2
SUBREQPCI-14/3 Creación de un diccionario (irreversibles)	IT1A-3
SUBREQPCI-14/4 Sobre el proceso de autenticación (irreversibles)	IT1A-4
SUBREQPCI-15/1 Distribución uniforme (criptográficos)	RC1B-1
SUBREQPCI-15/2 Permutación aleatoria (criptográficos)	RC1B-2
SUBREQPCI-15/3 Cambio de llave (criptográficos)	RC1B-3
SUBREQPCI-15/4 Cambio de PAN (criptográficos)	RC1B-4
SUBREQPCI-15/4 Cambio de PAN (criptográficos)	RC1B-4
Continúa en siguiente página	

Continuación	
Requerimiento	**Equivalente PCI**
SUBREQPCI-15/5 Verificación de la aleatoriedad (criptográficos)	RC1B-5
REQPCI-16 Almacenamiento de tokens (criptográficos)	RC1C
REQPCI-17 Seguridad de la administración de llaves (criptográficos)	RC4A
REQPCI-18 Sobre la longitud de las llaves (criptográficos)	RC4C
REQPCI-19 Independencia estadística (criptográficos)	RC4D
REQPCI-20 Generación y almacenamiento de tokens (no criptográficos)	RN1A
SUBREQPCI-21/1 Distribución equiprobable (no criptográficos)	RN1B-1
SUBREQPCI-21/2 Permutaciones aleatorias (no criptográficos)	RN1B-2
SUBREQPCI-21/3 Parámetros de tokenización (no criptográficos)	RN1B-3
SUBREQPCI-21/4 Verificación de la aleatoriedad (no criptográficos)	RN1B-5
REQPCI-22 Distribución imparcial (no criptográficos)	RN1C
REQPCI-23 Instancias estadísticamente independientes (no criptográficos)	RN1D
REQPCI-24 Proceso de detokenización (no criptográficos)	RN2A
REQPCI-25 Cifrado de la base de datos (no criptográficos)	RN3A
REQPCI-26 Seguridad de la administración de llaves (no criptográficos)	RN4A

Tabla 3.3: Resumen de requerimientos del PCI SSC para los sistemas tokenizadores.

3.1.2. Estándares y recomendaciones

En esta sección se resumen los contenidos de los estándares y recomendaciones del NIST relacionados con los requerimientos planteados en la sección anterior (3.1.1).

Administración de llaves (Sección 3.1.2.1)

800-57 *Recommendation for Key Management.* Disponible en [31].

800-130 *A Framework for Designing Cryptographic Key Management Systems.* Disponible en [32].

Generación de llaves (Sección 3.1.2.2)

800-108 *Recommendation for Key Derivation Functions Using Pseudorandom Functions.* Disponible en [34].

800-133 *Recommendation for Cryptographic Key Generation.* Disponible en [35].

Generación de bits pseudoaleatorios (Sección 3.1.2.3)

800-90A Revision 1 *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* Disponible en [33].

3.1.2.1. Administración de llaves

Tipos de llaves En [31], el NIST divide a las llaves criptográficas en 19 categorías, las cuales surgen de la combinación de la naturaleza de la llave (simétrica, pública o privada) con la funcionalidad que se le da (firmas, autenticación, cifrado, entre otras). En la tabla 3.4 se muestran dichas categorías: se marca con ✓ las combinaciones que tienen sentido práctico; por ejemplo, para firmar (y posteriormente validar) solamente se ocupan las llaves de un esquema asimétrico (i. e. públicas y privadas).

Para firmas: llaves que son usadas en algoritmos asimétricos para generar firmas digitales con posibles implicaciones a largo plazo. Cuando son manejadas correctamente, este tipo de llaves está diseñado para proveer AUTENTICACIÓN DE ORIGEN, autenticación de integridad y soporte para el no repudio.

Para autenticación: son usadas para proporcionar garantías sobre la identidad de un fuente generadora (i. e. AUTENTICACIÓN DE ORIGEN).

Para cifrado de datos: usadas en esquemas simétricos para proveer de confidencialidad a la información (esto es, para cifrar la información). Como se tratan de algoritmos simétricos (o de llave secreta), la misma llave se usa para quitar la confidencialidad (para descifrar).

Como envoltura de otras llaves: también llamadas llaves cifradoras de llaves (*key-encrypting keys*), son usadas para proteger otras llaves usando algoritmos simétricos. Dependiendo del algoritmo con el que se use la llave, esta también puede ser usada para proveer de validaciones de integridad.

		Naturaleza		
		Simétrica	Pública	Privada
Funcionalidad	Para firmas		✓	✓
	Para autenticación	✓	✓	✓
	Para cifrado de datos	✓		
	Como envoltura de otras llaves	✓		
	Para generadores aleatorios	✓		
	Como llave maestra	✓		
	Para el transporte de llaves		✓	✓
	Para el acuerdo de llaves	✓	✓	✓
	Efímeras para el acuerdo de llaves		✓	✓
	De autorización	✓	✓	✓

Tabla 3.4: Clasificación de llaves criptográficas

Para generadores aleatorios: llaves usadas para generar números o bits aleatorios.

Como llaves maestra: una llave maestra simétrica se usa para derivar otras llaves simétricas (p. ej. llaves de cifrado, o de envoltura). También se conoce como llave de derivación de llaves (*key-derivation key*).

Para el transporte de llaves: llaves utilizadas en esquemas asimétricos para proteger la comunicación en un proceso de acuerdo de llaves. Se usan para proteger otras llaves (p. ej. de cifrado, de envoltura) o información relacionada (p. ej. VECTORES DE INICIALIZACIÓN).

Para el acuerdo de llaves: utilizadas en algoritmos de acuerdo de llaves para establecer otras llaves. Generalmente funcionan en plazos muy largos.

Efímeras para el acuerdo de llaves: llaves de muy corto plazo que son usadas una sola vez en un proceso de establecimiento de otras llaves. En algunos casos la llave efímera puede ser usada más de una vez, pero dentro de la misma transacción.

De autorización: usadas para proveer y verificar los privilegios de una entidad. En un esquema simétrico, la llave es conocida tanto por la entidad que provee acceso, como por la que desea acceder.

Usos de llaves En general, las llaves se deben de utilizar para un solo propósito: el uso de la misma llave para dos operaciones criptográficas puede debilitar la seguridad provista por ambas; al limitar el uso de una llave se limita el nivel de riesgo de que esta se vea comprometida; algunos usos de llaves interfieren unos con otros.

La regla anterior no se extiende a situaciones en donde el mismo proceso provea de múltiples servicios. Por ejemplo, cuando una sola firma digital provee de autenticación de integridad y de autenticación de origen, o cuando una sola llave simétrica es usada para cifrar y para autenticar en una sola operación.

Criptoperiodos Un criptoperiodo es el rango de tiempo durante el cual una llave es válida. Algunas de las funciones de los criptoperiodos son:

- Limitar el total de daños si una sola llave se ve comprometida.
- Limitar el uso de un algoritmo en particular (*particular* en el sentido de la instancia misma del algoritmo).
- Limitar el tiempo disponible para intentos de ataques sobre los mecanismos que protegen a la llave del acceso sin autorización.
- Limitar el periodo en el cual la información puede verse comprometida por revelaciones inadvertidas de llaves a entidades sin autorización.
- Limitar el tiempo disponible para ataque criptoanalíticos.

A continuación se enlistan los principales factores a tomar en cuenta al momento de establecer un criptoperiodo:

- La fuerza del mecanismo criptográfico (el algoritmo, la FUERZA EFECTIVA de la llave, el tamaño de bloque, el modo de operación, etc.).
- El entorno de operacion (p. ej. un lugar de acceso restringido, una terminal pública).
- El volumen de información transmitida, o el número de transacciones.
- La función de seguridad (cifrado de datos, firma digital, derivación de llaves, etc.).
- El método de entrada de la llave (por teclado, a nivel de sistema).
- El número de nodos en una red que comparten la misma llave.
- El número de copias de la llave distribuidas.
- Rotaciones de personal.
- La amenaza de los adversarios (¿de quién se está protegiendo la información?, ¿cuáles son sus capacidades?)
- La amenza de avances tecnológicos (nuevos límites para el problema del logaritmo discreto, computadoras cuánticas).

La duración de los criptoperiodos también debe tomar en cuenta cuáles son los riesgos de las actualizaciones de llaves. En general, entre más cortos sean los criptoperiodos, mejor; sin embargo, si los mecanismos de distribución de llaves son manuales y propensos a errores humanos, entonces el riesgo se invierte. En estos casos (en especial cuando se utiliza CRIPTOGRAFÍA FUERTE) resulta mejor tener pocas y bien controladas distribuciones manuales, a que estas sean frecuentes y mal controladas.

Las consecuencias de una exposición se miden de acuerdo a qué tan sensible es la información, qué tan críticos son los procesos protegidos, y qué tan alto es el costo de recuperación en caso de exposición. La sensibilidad se refiere al periodo de tiempo durante el cual la información debe estar protegida (5 segundos, 5 minutos, 5 horas o 5 años) y a las consecuencias potenciales en caso de pérdida de protección. En general, entre más sensible sea la información protegida, el criptoperiodo debe ser menor (sin llegar a que sea contraproducente).

Algunos otros factores a tomar en cuenta incluyen el uso de las llaves y el costo de actualizaciones. En cuanto al uso de las llaves, generalmente se hace distinción en cuanto a si la información protegida solamente se está transfiriendo, o si se va a almacenar; en general los criptoperiodos son más largos en caso de almacenamiento, dado el costo que puede representar el recifrado de toda una base de datos. Esto está relacionado con el costo de las actualizaciones: cuando el volumen de información protegida es demasiado grande o cuando esta se encuentra distribuida en distintos puntos geográficos, el costo de un cambio de llaves puede ser demasiado elevado.

La tabla 3.5 muestra las sugerencias del NIST en [31] en cuanto a la duración de los criptoperiodos según cada tipo de llave.

Estados de llaves y transiciones En la figura 3.2 se muestra un diagrama de estados con los posibles comportamientos de una llave criptográfica. Un criptoperiodo inicia al entrar en el estado «activo» (transición 4) y termina al llegar al estado «destruido». En el caso de esquemas asimétricos, las transiciones se aplican a ambos pares de llaves. Se debe llevar un registro de la fecha y hora (y en algunos casos de las razones) de cualquier cambio de estado.

Hay varias posibles razones por las que se puede llegar a un estado «suspendido»: la entidad dueña de una firma digital no se encuentra disponible; se sospecha que la integridad de la llave está comprometida, por lo que se pasa a este estado en lo que se investiga más en profundidad; entre otros. Una llave que está en este estado («suspendido») no debe ser usada bajo ninguna circunstancia.

Las llaves que se encuentran en el estado «inactivo» no deben ser usadas para aplicar protección criptográfica, pero en algunos casos, pueden ser usadas para procesar información protegida con criptografía. Por ejemplo, las llaves simétricas usadas para autenticación, cifrado de datos o envoltura de llaves, pueden ser usadas para procesar información hasta que acabe el periodo del emisor de la llave emisora.

Tipo de llave	**Criptoperiodo**	
	Periodo de uso de emisor (PE)	**Periodo de uso de receptor (PR)**
1.- Llave privada para firma	1 a 3 años	-
2.- Llave pública para verificación de firma	Depende del tamaño de la llave	
3.- Llave simétrica para autenticación	≤ 2 años	$\leq PE + 3$ años
4.- Llave privada para autenticación	1 a 2 años	
5.- Llave pública para autenticación	1 a 2 años	
6.- Llave simétrica para cifrado de datos	≤ 2 años	$\leq PE + 3$ años
7.- Llave simétrica como envoltura	≤ 2 años	$\leq PE + 3$ años
8.- Llave simétrica para generadores aleatorios	ver SP800-90	-
9.- Llave simétrica maestra	alrededor de 1 año	-
10.- Llave privada para transporte	≤ 2 años	
11.- Llave pública para transporte	1 a 2 años	
12.- Llave simétrica para acuerdo	1 a 2 años	
13.- Llave privada para acuerdo	1 a 2 años	
14.- Llave pública para acuerdo	1 a 2 años	
15.- Llave privada efímera para acuerdo	Solo 1 transacción	
16.- Llave pública efímera para acuerdo	Solo 1 transacción	
17.- Llave simétrica para autorización	≤ 2 años	
18.- Llave privada para autorización	≤ 2 años	
19.- Llave pública para autorización	≤ 2 años	

Tabla 3.5: Criptoperiodos sugeridos por tipo de llave

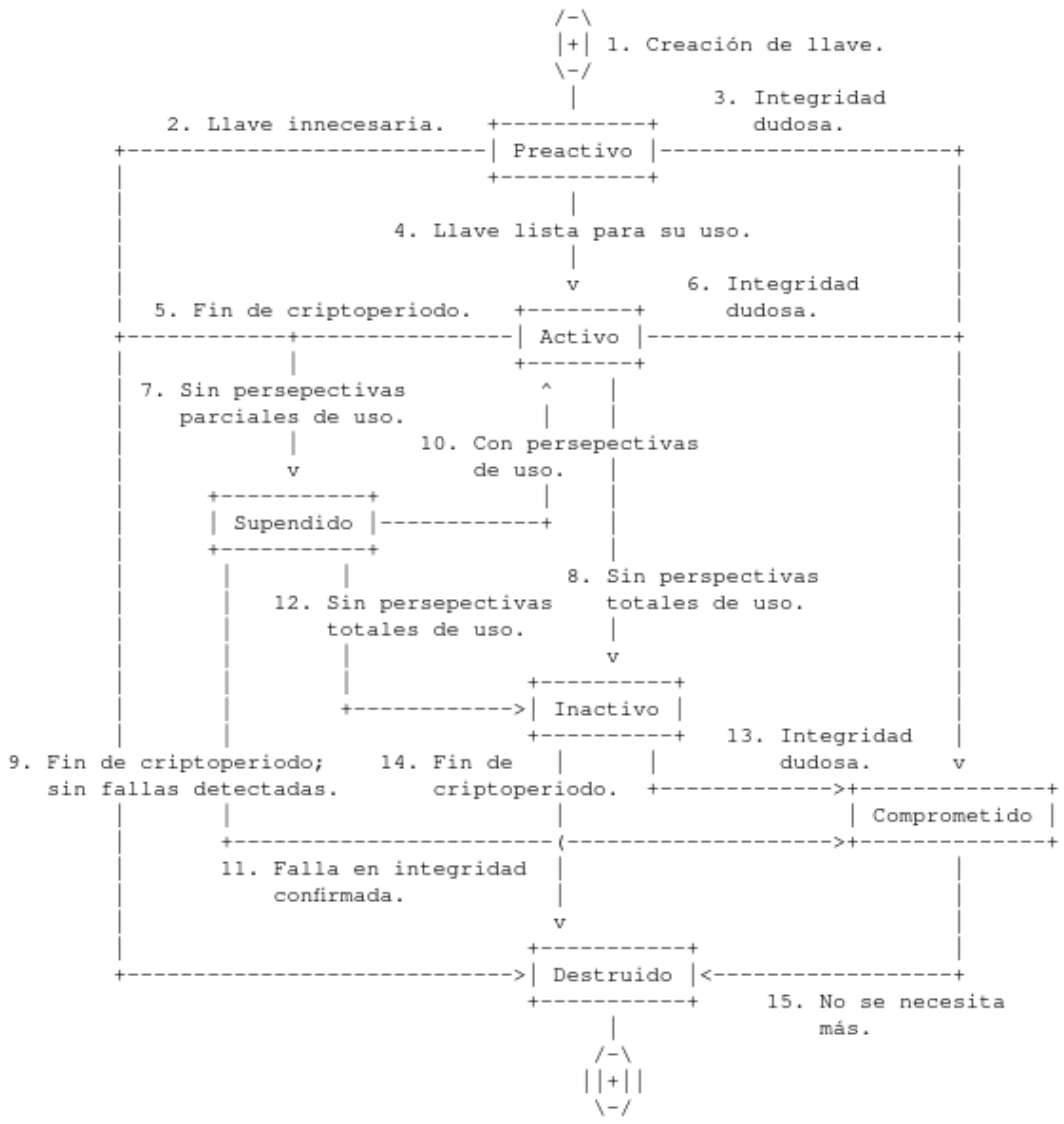


Figura 3.2: Diagrama de estado de llaves criptográficas.

El estado «comprometido» representa a las llaves a las que una entidad sin autorización tiene o ha tenido acceso. Las llaves comprometidas no deben ser usadas para aplicar protección criptográfica, sin embargo, en algunos casos es posible que sean usadas para procesar información (en condiciones sumamente controladas); por ejemplo, si la cierta información fue firmada antes de que se produjera la brecha de seguridad, las llaves pueden ser usadas para valida esta firma.

3.1.2.2. Generación de llaves

Generación de llaves en general

Métodos para la generación de llaves La generación de llaves se puede hacer por medio del uso de generadores de bits aleatorios (RBG), de la derivación de llaves a partir de otras, de la derivación de llaves a partir de una contraseña, y del uso de un esquema de acuerdo entre llaves o *key agreement*; pero todas, de forma directa o indirecta deben estar basadas en la salida de un RBG.

Donde generar las llaves La generación de llaves criptográficas debe ir de acuerdo con el FIPS 140 (descrito en [29]), y si es necesario que las llaves sean transferidas, se debe de hacer por medio de un canal seguro. Además, todos los valores aleatorios requeridos para la generación de las llaves deben ser generados dentro de un mismo módulo criptográfico.

Fuerza de la seguridad La fuerza de la seguridad de un método es una medición de la complejidad asociada a la recuperación de información secreta o de la seguridad relativa a un algoritmo criptográfico a partir de datos conocidos.

Se dice que un método soporta la fuerza de seguridad, si la fuerza de seguridad provista por dicho método es igual o mayor a la seguridad requerida para la protección de la información.

- La fuerza de seguridad para los RBG está basada en la ENTROPÍA o aleatoriedad que provee el mismo RBG.
- La seguridad de un algoritmo criptográfico se basa en el hecho de que las llaves que usan fueron generadas por medio de procesos que las proveyeron de una ENTROPÍA mayor o igual a la necesaria para el algoritmo, así como el mismo tamaño de las llaves.
- La fuerza de seguridad de una llave depende del algoritmo que la utilizará, su tamaño, el proceso con el que fue generada, y la forma en la que es utilizada.

Usos de las salidas de los RBG Suponiendo que K es una llave simétrica o un valor aleatorio que sirve como entrada de un algoritmo generador de pares de llaves asimétricas, K debe ser una cadena de bits tal que:

$$K = U \oplus V \tag{3.1}$$

donde U es una cadena de bits que se obtuvo de un RBG con el soporte de la fuerza de seguridad requerida, y V es una cadena de bits de la misma longitud, que además fue generada de manera tal que es independiente de U y viceversa. La independencia entre U y V se requiere debido a que se tiene que evitar que el conocimiento de alguna de estas cadenas pueda usarse para obtener información de la otra.

Generación de pares de llaves asimétricas Los pares de llaves solo pueden ser generados o por el propietario del las mismas llaves, o por un tercero de confianza capaz de proveerlas.

Las llaves privadas deben permanecer secretas dentro del módulo criptográfico del propietario o de un tercero de confianza, además de que, si se tienen que transferir dichas llaves, se debe de garantizar que solo el propietario o la parte generadora puedan verlas en claro.

Generación de llaves simétricas Las llaves deben de ser generadas por una o varias de las entidades que las compartirán, o por un tercero de confianza capaz de compartirlas de una manera segura.

Las llaves que son generadas directamente de un RBG deben de cumplir con la forma establecida en la ecuación 3.1.

Cuando sea necesario compartir una llave, su distribución debe de ser manual, o por medio de un envolvimiento de la misma o *key wrapping*, el cual debe de soportar la fuerza de seguridad requerida para la protección de la información que protege la llave.

Obtener llaves derivadas de una contraseña es una práctica cuestionable, debido a que comúnmente la aleatoriedad en las contraseñas es mínima, por tal razón al generarse llaves de esta manera, es fuertemente recomendado que los usuarios seleccionen contraseñas con una gran cantidad de ENTROPÍA. De cualquier manera, se considera que este tipo de llaves proveen una ENTROPÍA nula, a menos que la contraseña haya sido generada con un RBG.

Cuando se tiene un conjunto de llaves $K_1, \ldots, K_n$ generadas de forma independiente, estas pueden ser combinadas entre sí para formar una llave K. Igualmente, si se tiene un conjunto de bloques de información $V_1, \ldots, V_m$ que son independientes de sus respectivas llaves, se pueden combinar estos bloques y sus llaves para formar otra llave.

Los métodos aprobados para generar llaves a partir de la combinación de otras son:

- La concatenación de varias llaves. $K = K_1 \parallel \ldots \parallel K_n$.
- La aplicación de compuertas *xor* (o exclusiva) a varias llaves. $K = K_1 \oplus \ldots \oplus K_n$.
- La aplicación de compuertas *xor* (o exclusiva) a varias llaves y bloques de información. $K = K_1 \oplus \ldots \oplus K_n \oplus V_1 \oplus \ldots \oplus V_m$.

Cuando sea necesario reemplazar una llave, la nueva llave debe de ser completamente independiente de la anterior, para que el conocimiento de esta última, no proporcione ningún conocimiento de la nueva.

Funciones Pseudoaleatorias (PRF) Las funciones pseudoaleatorias o PRF (ver sección 3.1.2.3) son funciones computables en TIEMPO POLINOMIAL con un índice o SEMILLA s y una variable de entrada x, de manera que cuando s se selecciona aleatoriamente de S, es COMPUTACIONALMENTE INDISTINGUIBLE de una función aleatoria definida en el mismo dominio y rango que $PRF(s, x)$.

Cuando una llave criptográfica K_I es usada como la SEMILLA de una PRF, su salida se puede utilizar como MATERIAL DE LLAVES.

Para la derivación de llaves se tiene permitido el uso del KEYED-HASHED MESSAGE AUTHENTICATION CODE (HMAC) o del CMAC como función pseudoaleatoria.

Funciones de derivación de llaves (KDF) Las funciones de derivación de llaves o KEY DERIVATION FUNCTION (KDF) son funciones que a partir de una llave dada como entrada, pueden generar MATERIAL DE LLAVES capaz de ser empleado por varios algoritmos criptográficos.

Las llaves de entrada de este tipo de funciones son llamadas *key derivation keys* y deben ser llaves criptográficas generadas por medio de un RBG o por un proceso automático de establecimiento de llaves.

El MATERIAL DE LLAVES segmentado es usado como un conjunto de llaves criptográficas correspondientes a distintos algoritmos que pueden ofrecer diferentes servicios, por lo tanto las KDF deben de definir una manera de transformar este material en llaves distintas.

De forma general las KDF funcionan iterando n veces una función pseudoaleatoria para concatenar sus salidas hasta que se alcance la longitud de bits deseada para el MATERIAL DE LLAVES.

Modos de iteración Algo necesario para el funcionamiento de las KDF son los modos de iteración, ya que definen las entradas que se tendrán y el orden de los campos de salida en cada ciclo.

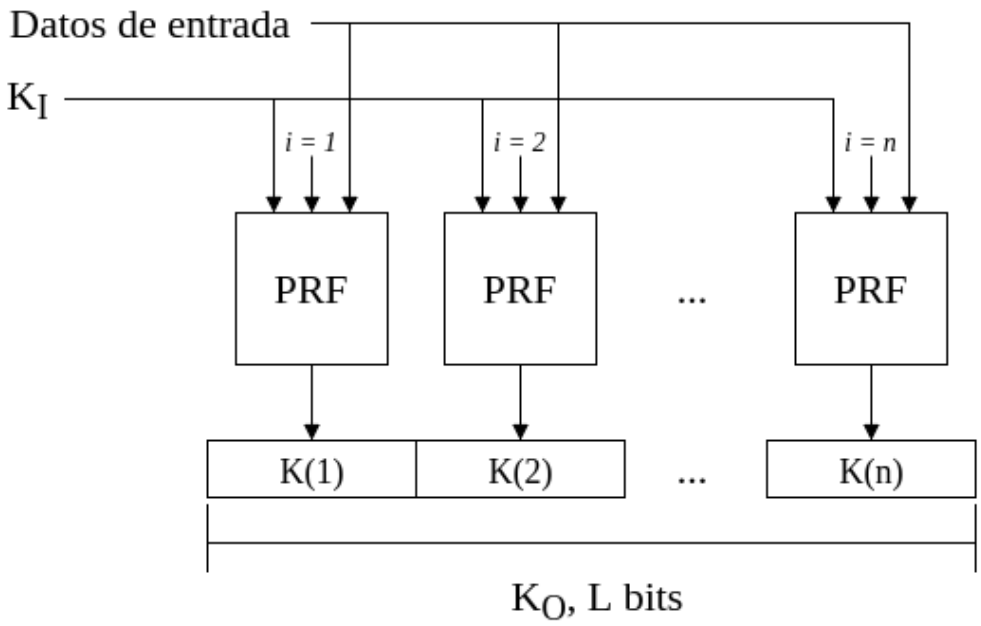


Figura 3.3: Diagrama del *counter mode*.

Counter mode En la figura 3.3 se aprecia la forma de operación de este modo de iteración, en el que los datos de entrada son concatenados en una cadena binaria de la forma: $Label \parallel 0x00 \parallel Context \parallel [L]_2$ donde el *Label* es un valor que identifica el propósito del MATERIAL DE LLAVES, $0x00$ es un octeto de ceros, el *Context* es una cadena con la información relacionada al MATERIAL DE LLAVES, y $[L]_2$ es la longitud del MATERIAL DE LLAVES que se desea obtener representada en binario. Como se observa, en este modo de iteración el contador forma parte los datos de entrada de la PRF.

Feedback mode Este modo de iteración opera tomando la salida de la PRF en la iteración anterior como parte los datos de entrada de la iteración actual. Cabe resaltar que como se muestra en la figura 3.4, es opcional tener un contador concatenado a estos datos, que son iguales que en el modo de operación anterior, solo que con un VECTOR DE INICIALIZACIÓN IV.

Double pipeline mode Como su nombre lo indica y a diferencia de los otros dos modos de iteración mencionados, este usa dos flujos, donde un flujo es el encargado de generar valores secretos, y el otro usa como entradas las salidas del primero para obtener K_O. Los datos de entrada son iguales que en el primer modo de iteración mencionado, y su funcionamiento está descrito en la figura 3.5.

Jerarquía de llaves El MATERIAL DE LLAVES proveniente de una derivación puede ser usado una o más veces como llave de entrada de otras derivaciones subsecuentes, así, es posible establecer una jerarquía

```
entrada:  La llave K_I, la longitud L y los valores de Label y Context.
salida:   K_O, con una longitud de L bits.
inicio
  calcular n = [L/h], donde h es la longitud de salida de la PRF.
  si n > 2^r − 1, indicar error y parar; donde r ≤ 32.
  resultado(0) = ∅
  para i = 1 hasta n:
    K(i) = PRF(K_I, [i]_2 ‖ Label ‖ 0x00 ‖ Context ‖ [L]_2).
    resultado(i) = resultado(i − 1) ‖ K(i).
  K_O = los L bits más a la izquierda del resultado(n).
fin
```

Pseudocódigo 3.1: Funcionamiento del *counter mode*.

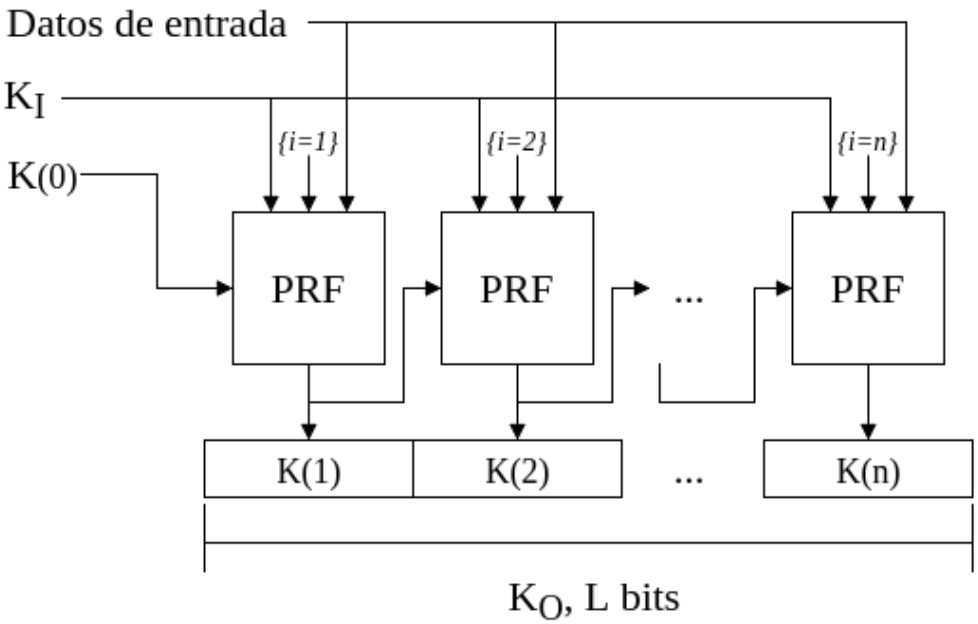


Figura 3.4: Diagrama del *feedback mode*.

entrada: La llave K_I, la longitud L, el vector de inicialización IV
y los valores de *Label* **y** *Context*.
salida: K_O, con una longitud de L bits.
inicio
  calcular $n = [\frac{L}{h}]$, donde h es la longitud de **salida** de la PRF.
  si $n > 2^{32} - 1$, indicar error **y** parar.
  $resultado(0) = \emptyset$.
  $K(0) = IV$.
  para $i = 1$ hasta n:
    $K(i) = PRF(K_I, K(i-1) \{\| [i]_2\} \| Label \| 0x00 \| Context \| [L]_2)$.
    $resultado(i) = resultado(i-1) \| K(i)$.
  $K_O =$ los L bits más a la izquierda del $resultado(n)$.
fin

Pseudocódigo 3.2: Funcionamiento del *feedback mode*.

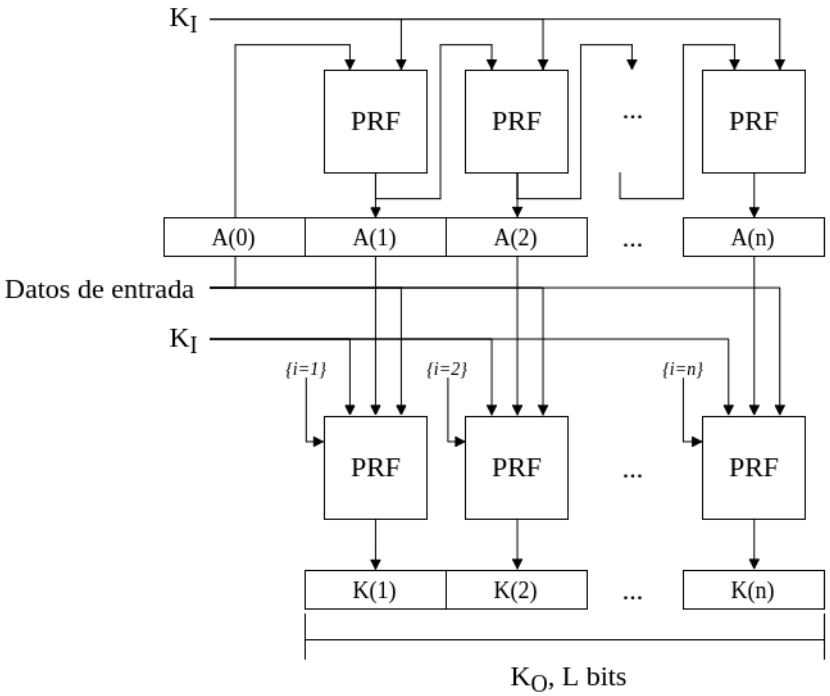


Figura 3.5: Diagrama del *double pipeline mode*.

```
entrada:  La llave K_I, la longitud L y los valores de Label y Context.
salida:   K_O, con una longitud de L bits.
inicio
  calcular n = [L/h], donde h es la longitud de salida de la PRF.
  si n > 2^32 - 1, indicar error y parar.
  resultado(0) = ∅.
  A(0) = IV = Label || 0x00 || Context || [L]_2
  para i = 1 hasta n:
    A(i) = PRF(K_I, A(i-1)).
    K(i) = PRF(K_I, A(i) {|| [i]_2} || Label || 0x00 || Context || [L]_2).
    resultado(i) = resultado(i-1) || K(i).
  K_O = los L bits más a la izquierda del resultado(n).
fin
```

Pseudocódigo 3.3: Funcionamiento del *double pipeline mode*.

en la que las llaves tienen distintos niveles.

Consideraciones de seguridad

Fuerza criptográfica La fuerza de seguridad de una KDF es medida por la cantidad de trabajo requerido para distinguir su cadena de salida de una cadena de bits que en verdad tenga una DISTRIBUCIÓN UNIFORME y cuente con la misma longitud, suponiendo que la llave de entrada K_I, es la única entrada desconocida para la KDF.

La longitud de la llave de entrada En algunas KDF el tamaño de la llave K_I está definido por la PRF que usan internamente, por ejemplo cuando se usa el CMAC, pero igualmente, otras KDF pueden usar llaves de cualquier tamaño, como al usar el HMAC como PRF.

Transformación de material de llaves a llaves criptográficas. La longitud en bits del MATERIAL DE LLAVES está limitada tanto por el algoritmo relacionado a la salida de la KDF, como por el modo de iteración que se usa.

Codificación de los datos de entrada La información de entrada de una KDF consiste en diferentes campos de datos, estos deben de estar ordenados de forma específica y sin ambigüedad, de manera que se tenga un método de codificación capaz de mapear cada campo individualmente. Esto es necesario para poder detectar ataques a la KDF que dependan de la manipulación de esta información.

Separación entre llaves Las llaves provenientes del MATERIAL DE LLAVES deben de estar separadas criptográficamente, de tal manera que no se comprometa la seguridad de ninguna de estas llaves derivadas. Cuando el MATERIAL DE LLAVES se obtiene de múltiples ejecuciones de la KDF usando la misma llave de entrada K_I, la KDF debe de garantizar que el MATERIAL DE LLAVES de una ejecución no comprometa al de cualquier otra.

Enlace de contexto Todo el MATERIAL DE LLAVES debe estar ligado a todas las entidades relacionadas para poder evitar errores de protocolo.

3.1.2.3. Generación de bits pseudoaleatorios

Existen dos maneras de generar bits aleatorios: la primera es producir bits de manera no determinística, donde el estado de cada uno (uno o cero) está determinado por un proceso físico impredecible. Estos generadores de bits aleatorios (RBG) son conocidos como generadores no determinísticos, o NON-DETERMINISTIC RANDOM BIT GENERATOR (NRBG). La otra manera, que será explorada a continuación, es calcular determinísticamente los bits mediante un algoritmo; estos generadores determinísticos son conocidos como DETERMINISTIC RANDOM BIT GENERATOR (DRBG).

Un DRBG tiene un mecanismo que utiliza un algoritmo que produce una secuencia de bits partiendo de un valor inicial que es determinado por una SEMILLA que, a su vez, está determinada por la salida de la fuente de aleatoriedad. Una vez que se tiene la SEMILLA y se determina el valor inicial, el DRBG es instanciado y puede producir valores. Dado a su naturaleza determinística, se dice que los valores producidos por el DRBG son pseudoaleatorios y no aleatorios; si la SEMILLA es mantenida oculta y el algoritmo fue bien diseñado, los bits de salida del DRBG serán impredecibles.

La entrada de ENTROPÍA es provista a un mecanismo DRBG para obtener una SEMILLA utillizando una fuente de aleatoriedad. La entrada de ENTROPÍA y la SEMILLA deben mantenerse secretas; que estos valores permanezcan secretos es una de las bases de la seguridad del DRBG. Otras entradas, como un NONCE o una cadena de personalización pueden ser utilizadas como entradas; estas pueden o no requerir ser mantenidas secretas también, y ser utilizadas para crear la SEMILLA inicial para el DRBG.

El estado interno es la memoria del DRBG y consiste en todos los valores que requiere el mecanismo (parámetros, variables, etcétera).

El mecanismo DRBG requiere cinco funciones; estas son explicadas con más detalle abajo:

1. Instanciación (*instantiate function*): obtiene la entrada de ENTROPÍA para crear una SEMILLA con la cual será creado un nuevo estado interno. La entrada puede ser combinada con una NONCE o una cadena de personalización.

2. Generación (*generate function*): genera bits pseudoaleatorios utilizando el estado interno actual; también tiene como salida un nuevo estado interno que es utilizado para el siguiente pedido.

3. Cambio de SEMILLA (*reseed function*): obtiene una nueva entrada de ENTROPÍA y la combina con el estado interno actual para crear una nueva SEMILLA y un nuevo estado interno.

4. Desinstanciación (*uninstantiate function*): elimina el estado interno actual.

5. Prueba de salud (*healt htest function*): determina que el mecanismo DRBG siga funcionando correctamente.

Cuando a un DRBG se le aplica la función de cambio de SEMILLA, es imperativo que la SEMILLA sea distinta a la que se utilizó en la función de instanciación. Cada SEMILLA define un nuevo periodo de SEMILLA (*seed period*) para la instanciación del DRBG. Una instanciación consiste en uno o más periodos de SEMILLA, estos comienzan cuando se obtiene una nueva SEMILLA y terminan cuando la siguiente SEMILLA es obtenida o el DRBG deja de utilizarse.

El estado interno deriva de la SEMILLA; este incluye el estado de trabajo (uno o más valores derivados de la SEMILLA que deben permanecer secretos, y la cuenta con el número de salidas que se han producido con esa semilla) y la información administrativa (el nivel de seguridad, etc). Es menester proteger el estado interno del DRBG. La implementación del mecanismo DRBG puede haber sido diseñado para tener múltiples instancias; en este caso, cada instancia debe tener su propio estado interno y el estado interno de una instancia DRBG jamás debe ser utilizado como estado interno para una instancia distinta. El estado interno no debe ser accesible a funciones distintas a las cinco del DRBG, ni a otras instancias del DRBG o a otros DRBG.

Los mecanismos especificados en [33] soportan cuatro niveles de seguridad: 112, 128, 192 y 256 bits. Este es uno de los parámetros que se necesitan para instanciar un DRBG; además, dependiendo de su diseño, cada mecanismo DRBG tiene sus restricciones de nivel de seguridad. El nivel de seguridad depende de la implementación del DRBG y la cantidad de ENTROPÍA que se da como entrada a la función de instanciación.

Los bits pseudoaleatorios obtenidos mediante un DRBG no deben ser utilizados por una aplicación que requiera un nivel mayor de seguridad que con el que fue instanciado el DRBG. La concatenación de dos salidas del DRBG tampoco proveen un nivel de seguridad más alto que del que fueron instanciados (por ejemplo, dos cadenas concatenadas de 128 bits no dan como resultado una cadena de 256 bits con el nivel de seguridad de 256 bits).

Semillas Las SEMILLAS deben ser obtenidas antes de generar bits pseudoaleatorios en el DRBG, pues esta es utilizada para instanciar al DRBG y determinar el estado inicial interno del mecanismo.

Cambiar la SEMILLA restaura el secreto de la salida del DRBG si el estado interno o la SEMILLA son conocidos. Hacer este cambio periódicamente es una buena manera de mantener a raya el peligro de que valores como la entrada de ENTROPÍA, la SEMILLA o el estado interno de trabajo; hayan sido comprometidos.

Los ingredientes para determinar una nueva SEMILLA para la función de instanciación son la entrada de entropía de una fuente aleatoria, un NONCE y una cadena de personalización (recomendada, pero no obligatoria). Para hacer un cambio de semilla, se necesita el estado interno actual, la entrada de ENTROPÍA y una entrada adicional opcional.

La longitud de la SEMILLA depende del mecanismo DRBG y el nivel de seguridad requerido; sin embargo, siempre debe ser de, al menos, el mismo número de bits de ENTROPÍA requerida.

La entrada de ENTROPÍA y la SEMILLA resultante deben de ser protegidas con el mismo cuidado con el que se protege la salida del DRBG; por ejemplo, si el mecanismo es utilizado para generar llaves, estos valores deben protegerse con la misma seguridad como son protegidas las llaves generadas. Además, la seguridad del DRBG depende de mantener en secreto la entrada de ENTROPÍA, por lo que esa entrada debe ser tratada como un parámetro crítico de seguridad (CRITICAL SECURITY PARAMETER (CSP)) y ser obtenido desde un módulo criptográfico que contenga la función necesaria o ser transmitido desde un canal seguro.

Cuando se requiera un NONCE para la construcción de la SEMILLA, esta debe cumplir con una de las siguientes dos condiciones: tener $nivel\_seguridad/2$ bits de ENTROPÍA o un valor que se espera no se repita más de lo que se repetiría una cadena aleatoria de $nivel\_seguridad/2$ bits. Aunque no debe ser mantenido en secreto, cada NONCE debe ser considerado como un CSP y debe ser único en el módulo criptográfico en donde se realiza la instanciación. El NONCE puede estar compuesto por uno o más de los siguientes valores:

1. Valor aleatorio generado para cada NONCE por un generador de bits aleatorios aprobado.
2. Una marca de tiempo con la resolución suficiente para que sea distinto cada vez que sea utilizado.
3. Un número de secuencia que se incremente constantemente.
4. Una combinación de una marca de tiempo y un número de secuencia que se incremente constantemente; tal que el número de secuencia regrese a su valor inicial solo cuando la marca de tiempo cambie.

Generar demasiadas salidas partiendo de una misma SEMILLA puede proveer suficiente información para ser capaz de predecir las salidas futuras; por lo que el cambio de SEMILLAS reduce riesgos de seguridad. Las SEMILLAS tienen una vida finita que depende el mecanismo DRBG utilizado. Es imperativo que las implementaciones respeten el límite de la vida de las SEMILLAS especificado para el mecanismo; y, cuando se alcance el límite de la vida de una SEMILLA, el DRBG no debe generar salidas hasta que se haya cambiado la SEMILLA o se cree una nueva instancia del DRBG (aunque se prefiere que se cambie la SEMILLA). Una SEMILLA jamás debe ser utilizada para inicializar o cambiar la semilla de otra instancia del DRBG o la

suya.

La cadena de personalización (que es opcional pero recomendada) es utilizada para derivar la SEMILLA, puede ser obtenida dentro o fuera del módulo criptográfico y hasta puede ser una cadena vacía, pues el DRBG no depende de esta cadena para obtener ENTROPÍA. De hecho, que el adversario conozca la cadena de personalización no disminuye el nivel de seguridad de una instancia de DRBG siempre y cuando la entrada de ENTROPÍA se mantenga desconocida. Esta cadena no es considerada un CSP; puede introducir datos adicionales al DRBG, tales como identificadores de usuario, aplicación, versiones, protocolos, marcas de tiempo, direcciones de red, números de serie, etcétera.

Funciones Un DRBG necesita cinco funciones para poder funcionar correctamente.

Instanciación Antes de generar bits pseudoaleatorios, el DRBG debe ser instanciado; esta función se encarga de revisar que los parámetros de entrada sean válidos, determina el nivel de seguridad para la instancia de DRBG que se generará, obtiene la entrada de ENTROPÍA capaz de soportar el nivel de seguridad y el NONCE (solo si es requerido), determina el estado interno inicial y, si se tienen varias instancias del DRBG simultáneas, obtiene un manejador de estado. Las entradas de la función son las siguientes:

1. Nivel de seguridad requerido.
2. Bandera que indica si se necesita o no la resistencia de predicción.
3. Cadena de personalización.
4. Entrada de ENTROPÍA.
5. NONCE

Las primeras entradas deben ser provistas por la aplicación consumidora. La salida de la función de instanciación a esta aplicación consiste en:

1. El estado del proceso; regresa un *EXITOSO* o un estado inválido indicando el error que hubo al instanciar al DRBG.
2. El manejador de estado, utilizado para identificar el estado interno de esta instancia para generar, cambiar la SEMILLA y demás funciones.

El algoritmo de la función de instanciación se puede observar en el pseudocódigo 3.4.

Cambio de semilla Cambiar la SEMILLA de una instancia no es requerido, pero es recomendado que se realice este proceso cada que sea posible. Cambiar la SEMILLA puede:

- Ser solicitado expresamente por la aplicación consumidora.
- Realizado cuando la aplicación consumidora requiere resistencia de predicción.

```
1  entrada:  nivel_seguridad_requerido, bandera_prediccion, cadena_personalizacion
2            entrada_entropia, nonce
3  salida:   estado, manejador_estado
4  inicio
5    si nivel_seguridad_requerido > mayor_nivel_seguridad_soportado:
6      regresar BANDERA_ERROR, invalido
7    si bandera_prediccion =verdadero Y resistencia_prediccion no es soportado:
8      regresar BANDERA_ERROR, invalido
9    si longitud(cadena_personalizacion) > longitud_maxima:
10     regresar BANDERA_ERROR, invalido
11   Asignar a nivel_seguridad el nivel más bajo de seguridad mayor o igual
12     a nivel_seguridad_requerido del conjunto {112, 128, 192, 256}
13   (estado, entrada_entropia) = obtener_entropia(nivel_seguridad, ...
14     ... longitud_min, longitud_max, bandera_prediccion)
15   si (estado ≠ EXITOSO):
16     regresar estado, invalido
17   Obtener nonce
18   estado_trabajo_inicial = INSTANCIAR_ALGORITMO(entrada_entropia, ...
19     ... nonce, cadena_personalizacion, nivel_seguridad)
20   Obtener manejador_estado para el estado interno vacío.
21   si no se encuentra un estado interno vacío:
22     regresar BANDERA_ERROR, invalido
23   Configurar el estado interno de la nueva instancia con los valores iniciales
24     y la información administrativa.
25   regresar (EXITOSO, manejador_estado)
26 fin
```

Pseudocódigo 3.4: DRBG, instanciación.

- Disparada por la función generadora cuando se alcanza un número predeterminado de salidas generadas.
- Disparada por eventos externos.

La función se encarga de revisar la validez de los parámetros de entrada, obtiene la entrada de ENTROPÍA de una fuente de aleatoriedad y, mediante el algoritmo de cambio de SEMILLA, combina el estado de trabajo actual con la nueva entrada de ENTROPÍA y valores adicionales para determinar el nuevo estado de trabajo actual. Las entradas para la función de cambio de SEMILLA son las siguientes:

1. El manejador de estado.
2. Bandera que indica si se requiere o no la resistencia de predicción.
3. Entradas adicionales (opcionales).
4. Entrada de ENTROPÍA.
5. Valores del estado interno e información administrativa.

Las primeras tres entradas deben ser provistas por la aplicación consumidora. La salida consiste en lo siguiente:

1. Estado que regresa la función.
2. Nuevo estado de trabajo interno.

El proceso para cambiar la SEMILLA se observa en el pseudocódigo 3.5.

Generación Esta función es utilizada para generar bits pseudoaleatorios después de haber utilizado la función de instanciación o de cambio de SEMILLA. Se encarga de validar los parámetros de entrada, llamar a la función de cambio de SEMILLA cuando sea requerida ENTROPÍA extra, generar los bits pseudoaleatorios, actualizar el estado de trabajo y regresar los bits a la aplicación consumidora que los pidió. La función tiene las siguientes entradas:

1. Manejador de estado.
2. El número de bits que se requieren.
3. El nivel de seguridad que se necesita.
4. Si debe ser resistente a predicción o no.
5. Entradas adicionales.
6. El estado de trabajo actual e información administrativa.

Después de haber sido generado, la salida consiste en lo siguiente

1. Estado.
2. Bits pseudoaleatorios.
3. Nuevo estado de trabajo.

```
entrada:  bandera_prediccion, manejador_estado, entrada_adicional
          entrada_entropia, estado_interno
salida:   estado, estado_trabajo_interno
inicio
  si manejador_estado indica un estado inválido o sin uso:
    regresar BANDERA_ERROR
  si se requiere resistencia de predicción y bandera_prediccion =falso:
    regresar BANDERA_ERROR
  si longitud(entrada_adicional) > longitud_max_entrada_adicional:
    regresar BANDERA_ERROR
  (estado, entrada_entropia) = obtener_entropia(nivel_seguridad, ...
    ... longitud_min, longitud_max, bandera_prediccion)
  si (estado ≠ EXITOSO):
    regresar estado
  estado_trabajo_nuevo = ALGORITMO_CAMBIAR_SEMILLA(estado_trabajo, ...
    ... entrada_entropia, entrada_adicional)
  Reemplazar el valor de estado_trabajo con estado_trabajo_nuevo
  regresar (EXITOSO)
fin
```

Pseudocódigo 3.5: DRBG, cambio de semilla.

El proceso para generar bits se puede observar en el pseudocódigo 3.6. Hay que tomar en cuenta cuando la implementación no tiene la capacidad de hacer el cambio de SEMILLA en el algoritmo: se quitan los pasos 6 y 7, y, si el estado indica que se necesita hacer el cambio, se regresa un error que indique que el DRBG no puede seguir siendo utilizado y hay que eliminar la instancia. También se debe tener en mente cuando se termina el ciclo de vida de la SEMILLA: cada que se llama al algoritmo generador, se revisa si el contador ha alcanzado el valor máximo que se encuentra en el estado interno; en caso de ser así, la función debe avisar que se requiere un cambio de SEMILLA.

Desinstanciación Esta función se encarga de liberar el estado interno de una instancia al borrar el contenido de su estado interno. La función requiere del manejador de estado de la instancia que se va a eliminar y regresa el estado de la función. El proceso para eliminar una instancia se puede observar en el pseudocódigo 3.7.

Mecanismos basados en funciones hash Los mecanismos DRBG pueden estar basados en funciones hash de un solo sentido; dos mecanismos basados en estas funciones son los HASH_DRBG y HMAC_DRBG. El nivel de seguridad que puede soportar cada uno es el nivel de resistencia a la PREIMAGEN que tiene la función hash (véase sección 2.4).

El mecanismo HASH_DRBG requiere el uso de una misma función hash en las funciones de instan-

```
1  entrada: bandera_prediccion, manejador_estado, entrada_adicional
2           nivel_seguridad_requerido, no_bits_requeridos, estado_interno
3  salida: estado, estado_trabajo, bits_pseudoaleatorios
4  inicio
5    si manejador_estado indica un estado inválido o sin uso:
6      regresar (BANDERA_ERROR, NULL)
7    si no_bits > no_bits_maximo
8      regresar (BANDERA_ERROR, NULL)
9    si nivel_seguridad_requerido > nivel_seguridad indicado en el estado interno:
10     regresar (BANDERA_ERROR, NULL)
11   si longitud(entrada_adicional) > longitud_max_entrada_adicional:
12     regresar (BANDERA_ERROR, NULL)
13   si se requiere resistencia de predicción y bandera_prediccion = falso:
14     regresar (BANDERA_ERROR, NULL)
15   Limpiar bandera_cambio_semilla_necesario
16   si bandera_cambio_semilla_necesario o bandera_prediccion están puestas:
17     estado = CAMBIO_SEMILLA(manejador_estado, prediccion_resistencia, entrada_adicional)
18     si estado ≠ EXITOSO:
19       regresar (estado, NULL)
20     Obtener el nuevo estado interno
21     entrada_adicional = NULL
22     Poner en cero bandera_cambio_semilla_necesario
23   (estado, bits_pseudoaleatorios, nuevo_estado_trabajo) = ALGORITMO_GENERAR(estado_trabajo, ...
24     ... no_bits_requeridos, entrada_adicional)
25   si estado indica que se requiere cambiar la semilla antes de poder generar bits:
26     Activar bandera_cambio_semilla_necesario
27     si se requiere resistencia de predicción:
28       Activar bandera_prediccion
29     Regresar al paso 7.
30   Reemplazar el estado_trabajo con nuevo_estado_trabajo.
31   regresar (EXITOSO, bits_pseudoaleatorios)
32 fin
```

Pseudocódigo 3.6: DRBG, generación.

```
1  entrada:  manejador_estado
2  salida:   estado
3  inicio
4    si manejador_estado indica un estado inválido:
5      regresar (BANDERA_ERROR)
6    Eliminar los contenidos del estado interno indicados por manejador_estado
7    regresar (EXITOSO)
8  fin
```

Pseudocódigo 3.7: DRBG, desinstanciación.

ciación, cambio de SEMILLA y generación. El nivel de seguridad de la función hash a utilizar debe igualar o ser mayor al nivel que se requiere por la aplicación consumidora del DRBG. En cambio, el mecanismo HMAC_DRBG utiliza múltiples ocurrencias de una función hash con llave; la misma función hash debe ser utilizada a lo largo del proceso de instanciación. Al igual que HASH_DRBG, la función hash debe tener, al menos, el mismo nivel de seguridad que la aplicación consumidora requiere para la salida del DRBG.

Mecanismos basados en cifradores por bloque Un cifrado por bloques DRBG está basado en un algoritmo de cifrado por bloques. La seguridad que puede alcanzar el DRBG depende del cifrado por bloques y el tamaño de llave utilizado. Se tiene al mecanismo CTR_DRBG, que utiliza un cifrado por bloques aprobado con el modo de operación de contador (véase sección 2.2.10); debe utilizarse el mismo algoritmo de cifrado y tamaño de llave para todas las operaciones de cifrado por bloques en el DRBG. El CTR_DRBG tiene una función actualizadora que es llamada por los algoritmos de instanciación, generación y cambio de semilla para ajustar el estado interno cuando hay nueva ENTROPÍA, se le dan entradas adicionales o cuando se actualiza el estado interno después de generar bits pseudoaleatorios.

Garantías Los usuarios de un DRBG requieren una garantía de que el generador en verdad produce bits pseudoaleatorios, que el diseño, implementación y uso de servicios criptográficos son adecuados para proteger la información del usuario y, finalmente, necesita una garantía de que el generador sigue operando correctamente. La implementación debe ser validada por un laboratorio acreditado por NATIONAL VOLUNTARY LABORATORY ACCREDITATION PROGRAM (NVLAP) para tener la certeza de que el mecanismo está bien implementado. Se requiere, además, para garantizar el funcionamiento del mecanismo, lo siguiente:

Documentación mínima se debe proveer, al menos, el siguiente conjunto de documentos; una gran parte podrían pertenecer al manual de usuario:

1. Documentación del método para obtener la entrada de entropía.
2. Documentar cómo la implementación fue diseñada para permitir la validación de la implementación y revisar su estado.

3. Documentar el tipo de mecanismo DRBG y las primitivas criptográficas utilizadas.
4. Documentar los niveles de seguridad soportados por la implementación.
5. Documentar las características que soporta la implementación.
6. Si las funciones del DRBG están distribuidas, especificar los mecanismos que se usan para proteger la confidencialidad e integridad de las partes del estado interno que son trasnferidas entre las partes distribuidas del DRBG.
7. Indicar si se utiliza una función de derivación; si no se utiliza, documentar que la implementacion solo puede utilizarse cuando la entrada de ENTROPÍA completa está disponible.
8. Documentar todas las funciones de soporte.
9. Si se requieren hacer pruebas periódicas para la función generadora, documentar los intervalos y justificarlos.
10. Documentar si las funciones del DRBG pueden ser puestas a prueba sobre demanda.

Pruebas de validación de la implementación El mecanismo DRBG debe ser diseñado para ser probado y poder asegurar que el producto está correctamente implementado. Debe proveerse una interfaz para realizarle pruebas que permita insertar los datos de entrada y obtener los datos de salida. Todas las funciones que se incluyen en la implementación deben ser probadas en la funcionalidad de las pruebas de salud.

Pruebas de salud La implementación DRBG debe realizarse pruebas de salud a sí mismo para asegurarse de que continua operando correctamente. Se deben realizar pruebas del tipo *known answer*, donde se tiene una entrada para la que ya se sabe la respuesta correcta.

Manejo de errores Se indica para cada función del mecanismo qué errores son los esperados; es menester indicar el tipo de error que ocurrió.

3.1.3. Pruebas para generadores de números aleatorios y pseudoaleatorios

3.1.3.1. Definiciones

Existen dos tipos básicos de generadores, los RANDOM NUMBER GENERATOR (RNG) y los PRNG.

Generadores aleatorios Este tipo de generador usa fuentes de ENTROPÍA no determinísticas para producir números aleatorios, dichas fuentes de ENTROPÍA normalmente consisten en la medición de eventos físicos, como el ruido en un circuito eléctrico.

La salidas de los RNG puede usarse directamente para obtener números aleatorios, o como una entrada de un PRNG. En el primer caso, es necesario que la salida satisfaga pruebas estadísticas para determinar que dicha salida parece aleatoria.

Los RNG que se usan con fines criptográficos tienen que ser impredecibles (ver sección 3.1.3.1), por lo cual es recomendado combinar varias fuentes de ENTROPÍA, dado que algunas (como los vectores de tiempo) son demasiado predecibles. De cualquier forma, en la mayoría de los casos es mejor usar los PRNG para obtener de manera directa números aleatorios.

Generadores pseudoaleatorios Estos son generadores de múltiples números aleatorios a partir de una o varias entradas llamadas SEMILLA. Cuando es necesario que la salida sea impredecible, la SEMILLA debe ser aleatoria y también impredecible por sí sola, razón por lo cual esta se debe de obtener por medio de un RNG.

La salida de los PRNG normalmente son funciones determinísticas de la SEMILLA, por lo que la verdadera aleatoriedad queda en la generación de esta misma, motivo por el que se usa el término de *pseudoaleatorio* y lo que causa la necesidad de mantener en secreto la SEMILLA.

Las mejores cualidades de los PRNG son que, al contrario de lo que se creería por el nombre, suelen parecer más aleatorias que las RNG, así como que producen números aleatorios más rápido.

Aleatoriedad Una secuencia de bits aleatoria puede interpretarse como una secuencia en la que el valor de cada bit es independiente, y los valores que lo conforman están uniformemente distribuidos (ver DISTRIBUCIÓN UNIFORME).

Impredecibilidad Los generadores aleatorios y pseudoaleatorios deben de ser impredecibles. La impredecibilidad es la incapacidad de conocer la salida de un bit $n + 1$ aun conociendo los valores del bit 0 al n si la SEMILLA es desconocida. Esta también es la incapacidad de determinar la SEMILLA usando el conocimiento de los valores generados por la misma.

Hay que resaltar que conociendo tanto la SEMILLA como el algoritmo generador, el PRNG es completamente predecible, por lo cual en la mayoría de los casos el algoritmo es público y la SEMILLA se mantiene secreta.

Pruebas estadísticas Estas sirven para determinar si la secuencia a evaluar puede ser considerada una secuencia aleatoria. Existe una gran diversidad de pruebas estadísticas para comparar y evaluar una secuencia generada por un RNG o un PRNG contra una secuencia verdaderamente aleatoria, cada una de estas buscando la existencia o ausencia de patrones para poder indicar si la secuencia parece o no aleatoria.

Es necesario aclarar que ningún conjunto de pruebas puede ser considerado completo, y que el resultado de determinada prueba es complementado por otras, motivo por el cual los resultados obtenidos deben ser interpretados cuidadosamente para no llegar a conclusiones incorrectas.

En **nist˙pruebas** se determina que las pruebas estadísticas están formuladas para determinar si una secuencia binaria cumple con la hipótesis nula *(H0)*, o la hipótesis alternativa *(Ha)*, las cuales dicen que la secuencia probada es aleatoria y no aleatoria respectivamente. Cada prueba descrita en la sección 3.1.3.2 consiste en aceptar alguna de las dos hipótesis mencionadas.

Para cada prueba, es necesario elegir una medida estadística relevante de aleatoriedad, para así establecer una DISTRIBUCIÓN DE PROBABILIDAD y un valor crítico que sirvan de referencia, y durante la prueba, se pueda comparar el valor estadístico de la secuencia probada con el valor elegido; aceptando la hipótesis nula en caso de que no se exceda del valor crítico y rechazándola en caso contrario. De manera práctica, las pruebas estadísticas funcionan dado que la DISTRIBUCIÓN DE PROBABILIDAD de referencia y el valor crítico son seleccionados asumiendo que la secuencia es aleatoria.

Hay que mencionar que estas pruebas estadísticas pueden tener dos tipos de errores, que la secuencia sea aleatoria y se concluya que no lo es, *(error tipo I)*, y que la secuencia no sea aleatoria y se concluya que sí, *(error tipo II)*. La probabilidad de que ocurra el error de tipo I es normalmente llamada *nivel de significancia* de la prueba, y se denota como α, este valor dentro de la criptografía es común que ronde cerca de 0.01. La probabilidad de que ocurra el error de tipo II se denota con β y a diferencia de α, no es un valor fijo y es más difícil de calcular.

Uno de los principales objetivos de las pruebas es minimizar la probabilidad de β, por lo cual se selecciona un tamaño de muestra n y un valor para α, para después establecer un valor crítico que produzca el valor β más pequeño. Esto es posible gracias a que las probabilidades α y β están relacionadas entre sí y con el tamaño de muestra n.

Cada prueba se basa en un valor estadístico de prueba calculado, que es una función de los mismos datos, por lo cual si el valor estadístico de prueba es S y el valor crítico es t, entonces:

$$\alpha = P(S > t \,||\, H0\, es\, verdadera) = P(rechazar\, H0 \,||\, H0\, es\, verdadero) \tag{3.2}$$

$$\beta = P(S \leq t \,||\, H0\, es\, falso) = P(aceptar\, H0 \,||\, H0 es\, falso) \tag{3.3}$$

Las pruebas estadísticas calculan un valor P, que indica la probabilidad de que un RNG perfecto haya producido una secuencia menos aleatoria que la secuencia que se probó, dado el tipo de aleatoriedad evaluada por la prueba. En caso de que el valor de P sea de 1 en una prueba, se considera que la secuencia parece tener una aleatoriedad perfecta y en caso de que el valor sea 0, la secuencia parece ser por completo no aleatoria.

Eligiendo un nivel de significancia α para las pruebas, si $P \geq \alpha$, entonces la hipótesis nula es aceptada, es decir, la secuencia parece ser aleatoria; y si $P < \alpha$, entonces la hipótesis nula es rechazada, es decir, la secuencia parece ser no aleatoria. Normalmente el valor de α se elige en el rango de [0.001, 0.01].

3.1.3.2. Pruebas

El paquete de pruebas descrito en el estándar del NIST *800-22R1A: Statistical Suite for Random Number Generators*, disponible en **nist˙pruebas**, es un conjunto de 15 algoritmos que se encargan de probar la aleatoriedad de secuencias binarias de longitudes arbitrariamente largas producidas por un RNG o un PRNG.

Prueba de frecuencia *(The frequency (Monobit) Test)*

Esta prueba se encarga de medir la proporción de ceros y unos en toda la secuencia binaria. Su propósito es determinar si la cantidad de ceros y de unos es parecida a la que se esperaría en una secuencia verdaderamente aleatoria. Algo a tener en cuenta es que todos las pruebas subsecuentes dependen de pasar esta prueba.

Prueba de frecuencia en un bloque *(Frequency Test within a Block)*

Esta prueba se centra en medir la proporción de unos dentro de N bloques de M-bits de la secuencia a probar. El propósito de la prueba es saber si la frecuencia de unos se aproxima a $M/2$ como sería de esperar de una secuencia aleatoria.

Prueba de carreras *(The Runs Test)*

Esta prueba se encarga de calcular el número de carreras en la secuencia que se está probando, donde una carrera es una subsecuencia ininterrumpida de bits del mismo valor encerrada entre bits del valor opuesto. El objetivo de la prueba es determinar si el número de carreras en la secuencia es parecido al que se esperaría de una secuencia aleatoria, particularmente, establece si la oscilación entre ceros y unos es muy rápida o muy lenta. Cabe resaltar que la prueba tiene como requisito pasar la prueba de frecuencia.

Prueba de la carrera más larga en un bloque *(Tests for the Longest-Run-of-Ones in a Block)*

El propósito de esta prueba es determinar si la longitud de la carrera más larga en la secuencia es consistente con la longitud de la carrera más larga de una secuencia aleatoria. Se tiene que resaltar que encontrar una irregularidad en la carrera más larga de unos implica una irregularidad en la carrera más larga de ceros, por lo cual solo es necesario hacer la prueba para la carrera más larga de unos.

Prueba del rango de matriz binaria *(The Binary Matrix Rank Test)*

La prueba tiene como finalidad revisar si existen dependencias lineales entre subcadenas de longitud fija de la secuencia a probar.

Prueba Espectral *(The Discrete Fourier Transform (Spectral) Test)*

Esta prueba se encarga de obtener los picos más altos de la transformada discreta de Fourier de la secuencia. Su propósito es detectar comportamientos periódicos en la secuencia que puedan indicar que esta no puede asumirse como aleatoria. De manera más precisa, la prueba detecta si el número de picos que exceden un umbral del 95 % son significativamente diferentes al 5 %.

Prueba de coincidencia sin superposición *(The Non-overlapping Template Matching Test)*

La prueba tiene como finalidad detectar generadores que producen una gran cantidad de ocurrencias de un patrón no periódico en sus secuencias de salida. Para esta prueba se utiliza una ventana de m bits que recorre la secuencia, para buscar patrones específicos de m bits. Dado que esta prueba es sin traslape, en caso de que se encuentre el patrón se recorre la ventana m bits y de lo contrario se recorre uno.

Prueba de coincidencia con superposición *(The Overlapping Template Matching Test)*

La prueba tiene como finalidad detectar generadores que producen una gran cantidad de ocurrencias de un patrón no periódico en sus secuencias de salida. Para esta prueba se utiliza una ventana de m bits que recorre la secuencia, para buscar patrones específicos de m bits. Dado que esta prueba es con traslape, independientemente de si se encontra o no un patrón, la ventana se recorre un bit.

Prueba estadística universal de Maurer *(Maurer's "Universal Statistical" Test)*

Esta prueba se centra en medir el número de bits existentes entre patrones que coinciden, medida que se relaciona con el tamaño mínimo para comprimir una secuencia. La prueba tiene como finalidad determinar si una secuencia puede comprimirse de manera significativa sin perder información, lo cual significa que no es aleatoria.

Prueba de complejidad lineal *(The Linear Complexity Test)*

El objetivo de la prueba es determinar si la secuencia a probar es lo bastante compleja como para considerarse aleatoria, esto por medio de medir el tamaño de su REGISTRO DE DESPLAZAMIENTO CON RETROALIMENTACIÓN LINEAL correspondiente, ya que las secuencias aleatorias se caracterizan por tener valores altos en este parámetro.

Prueba serial *(The Serial Test)*

Esta prueba tiene como objetivo determinar si el número de ocurrencias de 2^m patrones de m bits son aproximadamente las mismas que se esperarían de una cadena aleatoria, considerando que al ser una secuencia aleatoria, es una secuencia uniforme, y cada patrón de m bits tiene la misma probabilidad de aparecer que los demás.

Prueba de entropía aproximada *(The Approximate Entropy Test)*

La prueba se encarga de comparar la frecuencia de sobreponer dos bloques de longitudes consecutivas con el resultado esperado de una secuencia que es aleatoria.

Prueba de sumas acumulativas *(The Cumulative Sums (Cusums) Test)*

Su objetivo es determinar si las sumas acumulativas de una secuencia parcial de la secuencia probada son muy grandes o pequeñas con relación a las sumas acumulativas de una secuencia aleatoria. Las sumas aculumativas son consideradas como caminatas aleatorias, y para las secuencias aleatorias la realización de una caminata aleatoria debe tener un valor cercano a cero.

Prueba de excursiones aleatorias *(The Random Excursions Test)*

Esta prueba tiene como finalidad determinar si el número de apariciones de un estado x en un ciclo es parecido al número que se esperaría de una secuencia aleatoria. Esta prueba está conformada de 8 subpruebas, cada una para los valores de estado x de -4, -3, -2, -1, 1, 2, 3, 4.

Prueba variante de excursiones aleatorias *(The Random Excursions Variant Test)*

La prueba se centra en calcular el número de veces que un estado en particular ocurre en una suma acumulativa de una camita aleatoria. Su propósito, es detectar desviaciones del número esperado de ocurrencias de varios estados en una caminata aleatoria. Esta prueba está conformada de 18 subpruebas, cada un para los valores de estado x de -9 a de -1 y 1 a 9.

3.1.4. Algoritmos implementados

En esta sección se documentan los algoritmos para generar TOKENS que serán implementados; cada apartado incluye una descripción del algoritmo y su pseudocódigo. Antes de la documentación, se presentan dos clasificaciones de los algoritmos tokenizadores: la primera es la propuesta por el PCI SSC y mencionada

en la sección 3.1.1; la segunda clasificación es propuesta por los autores de este documento, tomando en cuenta nociones criptográficas más estrictas.

Clasificación del PCI SSC

- TOKENS reversibles:
  - TOKENS criptográficos:
    1. BPS (véase sección 3.1.4.2).
    2. FFX (véase sección 3.1.4.1).
  - TOKENS no criptográficos:
    1. AHR (véase sección 3.1.4.4).
    2. TKR (véase sección 3.1.4.3).
    3. DRBG

Cabe resaltar que, de acuerdo con esta clasificación, los TOKENS generados mediante DRBG pueden quedar también en la clasificación de TOKEN no reversible, no autenticable. Se propone una nueva clasificación, pues esta no parece muy acertada; por ejemplo, del lado de los irreversibles se pueden generar TOKENS autenticables (o no autenticables) mediante funciones hash criptográficas, sin embargo, estos TOKENS no son *criptográficos*. Respecto a los reversibles, se toman por ejemplo, a los algoritmos TKR, que hace uso de PRIMITIVAS CRIPTOGRÁFICAS, y AHR, que hace uso de un cifrado por bloques y una función hash criptográfica; ninguno de los cuales queda en la categoría de los criptográficos, pues se guardan las relaciones PAN-TOKEN en una base de datos.

Clasificación propuesta

- Criptográficos
  - Reversibles:
    1. BPS (véase sección 3.1.4.2).
    2. FFX (véase sección 3.1.4.1).
  - Irreversibles:
    1. AHR (véase sección 3.1.4.4).
    2. TKR (véase sección 3.1.4.3).
    3. DRBG.
- No criptográficos:

1. NRBG.

La clasificación propuesta divide primeramente entre los que utilizan criptografía y los que no; se pone como ejemplo a los item NRBG como algoritmos no criptográficos, ya que su aleatoriedad depende de fenómenos físicos. Luego, dentro de los criptográficos, se divide a su vez entre algoritmos reversibles e irreversibles: los primeros, al igual que en el PCI SSC, son aquellos algoritmos que, dados el TOKEN y la llave, pueden regresar y obtener el PAN; los algoritmos pertenecientes a la segunda subcategoría (irreversible) son aquellos que necesitan guardar la relación PAN-TOKEN para poder obtener el PAN perteneciente a un TOKEN. Dado que, de acuerdo con el NIST, los DRBG utilizan funciones hash o cifradores por bloque, pertenecen también a la categoría de los criptográficos irreversibles.

3.1.4.1. Algoritmo FFX

FORMAT-PRESERVING FEISTEL-BASED ENCRYPTION (FFX) es un cifrador que preserva el formato presentado en [36] por Mihir Bellare, Phillip Rogaway y Terence Spies. En su forma más general, el algoritmo se compone de 9 parámetros que permiten cifrar cadenas de cualquier longitud en cualquier alfabeto; los autores también proponen dos formas más específicas (dos colecciones de parámetros) para alfabetos binarios y alfabetos decimales: A2 y A10, respectivamente.

FFX ocupa redes Feistel (ver sección 2.2.3) junto con PRIMITIVAS CRIPTOGRÁFICAS adaptadas (usadas como función de ronda) para lograr preservar el formato. La idea general para las posibles funciones de ronda es interpretar la salida binaria de una primitiva (p. ej. un cifrador por bloques, una función hash, un cifrador de flujo) de forma que tenga el formato deseado; esto es, que tenga la misma longitud y se encuentre en el mismo alfabeto que la entrada. Es importante notar que esta función no tiene por qué ser invertible: las redes Feistel ocupan la misma operación tanto para cifrar como para descifrar.

La operación general del algoritmo es la misma que la operación de una red Feistel. Para redes desbalanceadas:

$$\begin{aligned} L_i &= R_{i-1} \\ R_i &= F_k(R_{i-1}) \oplus L_{i-1} \end{aligned} \tag{3.4}$$

Y para redes alternantes:

$$\begin{aligned} L_i &= \begin{cases} F_k^1(R_{i-1}) \oplus L_{i-1}, & \text{si } i \text{ es par} \\ L_{i-1}, & \text{si } i \text{ es impar} \end{cases} \\ R_i &= \begin{cases} R_{i-1}, & \text{si } i \text{ es par} \\ F_k^2(L_{i-1}) \oplus R_{i-1}, & \text{si } i \text{ es impar} \end{cases} \end{aligned} \tag{3.5}$$

Parámetro	**Valor**	**Comentario**
radix	10	alfabeto de dígitos decimales
longitudes	$[4, 36]$	rango de cadenas aceptadas
llaves	$\{0,1\}^{128}$	misma longitud que para AES
tweaks	$\text{BYTE}^{\leq 2^{64}-1}$	cadena de longitud arbitraria
suma	por bloque	combinación por operaciones a nivel de bloque
método	alternante	redes Feistel alternantes
split	0	lo más cercano al centro posible
rondas	12, 18 o 24	depende de longitud de mensaje

Tabla 3.6: Colección de parámetros FFX A10.

Especificación de parámetros A continuación se espifican los 9 parámetros de FFX.

1. ***Radix***. Número que determina el alfabeto usado. $C = \{0, 1, \ldots, \text{radix} - 1\}$. Tanto el texto en claro como el texto cifrado pertenecen a este alfabeto.

2. **Longitudes**. El rango permitido para longitudes de mensaje.

3. **Llaves**. El conjunto que representa al espacio de llaves.

4. ***Tweaks***. El conjunto que representa al espacio de *tweaks*.

5. **Suma**. El operador utilizado en la red Feistel para combinar la parte izquierda con la salida de la función de ronda.

6. **Método**. El tipo de red Feistel a ocupar: desbalanceada o alternante.

7. ***Split***. El grado de desbalanceo de la red Feistel.

8. **Rondas**. El número de rondas de la red Feistel.

9. **F**. La función de ronda. Recibe la llave, el *tweak*, el número de ronda y un mensaje; regresa una cadena del alfabeto de la longitud apropiada.

FFX A10 De las dos colecciones de parámetros presentadas en [37], la que se adapta mejor al dominio de los números de tarjetas de crétdito es A10, por lo que es la única que se presenta aquí. En la tabla 3.6 se muestran los valores con los que la colección FFX A10 inicializa FFX para crear un cifrador en el dominio decimal.

La dependencia entre el número de rondas y la longitud del mensaje está dada por la siguiente relación

(n es la longitud del mensaje):

$$\text{rondas } = \begin{cases} 12 & \text{si } 10 \leq n \leq 36 \\ 18 & \text{si } 6 \leq n \leq 9 \\ 24 & \text{si } 4 \leq n \leq 5 \end{cases} \tag{3.6}$$

La función de ronda concatena el *tweak*, el mensaje de entrada y los demás parámetros usados por la red en una sola cadena; esta cadena se cifra con AES CBC MAC; la salida se parte en dos (mitad derecha y mitad izquierda) y es operada para que el número resultado tenga el mismo número de dígitos que la cadena de entrada. Esta última operación se describe con la sguiente ecuación ($y\prime$ es la parte izquierda y $y\prime\prime$ la derecha):

$$z = \begin{cases} y\prime\prime \mod 10^m, & \text{si } m \leq 9 \\ (y\prime \mod 10^{m-9}) \cdot 10^9 + (y\prime\prime \mod 10^m), & \text{en cualquier otro caso} \end{cases} \tag{3.7}$$

El valor de m corresponde al *split* en la ronda actual; esto es la longitud de la cadena de entrada.

3.1.4.2. Algoritmo *BPS*

La información que aquí se presenta puede ser consultada con mayor detalle en [38].

BPS es uno de los algoritmos de cifrado que preservan el formato existentes, y es capaz de cifrar cadenas de longitudes casi arbitrarias que estén formadas por cualquier conjunto de caracteres.

BPS está conformado por 2 partes fundamentales, un cifrado interno BC, encargado de cifrar bloques de longitud fija; y un modo de operación, encargado de extender la funcionalidad de el cifrador BC y permitir que *BPS* cifre cadenas de varias longitudes.

El cifrado interno BC El cifrado por bloques que usa *BPS* internamente se define como

$$BC_{F,s,b,w}(X, K, T) \tag{3.8}$$

Donde:

- F es un cifrador por bloques de f bits de salida, por ejemplo: TDES, AES, SHA-2.
- s es la cardinalidad del conjunto de caracteres del bloque a cifrar.
- b es la longitud del bloque a cifrar, cumpliendo con $b \leq 2 \cdot |log_s(2^{f-32})|$.
- w es el número (par) de rondas de la red Feistel interna (véase 2.2.3).

- X es la cadena o bloque de longitud b a cifrar.
- K es una llave acorde al cifrador por bloques F.
- T es un *tweak* de 64 bits.

Proceso de cifrado BC.

Para poder cifrar la cadena X:

1. Se tiene que dividir el *tweak* T de 64 bits en 2 sub*tweak*s T_L y T_R de 32 bits. Viendo a T como un número entero codificado en binario se puede calcular $T_R = T \mod 2^{32}$ y $T_L = (T - T_R)/2^{32}$

2. Igualmente, se tiene que dividir en 2 la cadena X para obtener las subcadenas X_L y X_R con una longitud l y r respectivamente. Dado que la longitud b de la cadena no siempre es par, se tiene que, si b es par, entonces tanto l como r son igual a $b/2$, pero en caso de que b sea impar, l va a ser igual a $(b+1)/2$ y r igual a $(b-1)/2$.

3. Partiendo de que el cifrador BC se compone de w rondas de una red Feistel, se define L_i y R_i (parte izquierda y parte derecha de la red en la i-ésima ronda), y se inicializan en:

$$L_0 = \sum_{j=0}^{l-1} X_L[j] \cdot s^j \tag{3.9}$$

$$R_0 = \sum_{j=0}^{r-1} X_R[j] \cdot s^j \tag{3.10}$$

4. Ahora por cada ronda $i < w$ y cifrando con el cifrador por bloques E.

   Si i es par:

$$L_{i+1} = L_i \boxplus E_K((T_R \oplus i) \cdot 2^{f-32} + R_i) \qquad (mod\ s^l) \tag{3.11}$$

$$R_{i+1} = R_i \tag{3.12}$$

   Si i es impar:

$$R_{i+1} = R_i \boxplus E_K((T_L \oplus i) \cdot 2^{f-32} + L_i) \qquad (mod\ s^r) \tag{3.13}$$

$$L_{i+1} = L_i \tag{3.14}$$

5. Por último se tiene que descomponer tanto a L_w como a R_w para obtener a Y_L y a Y_R respectivamente, las cuales concatenadas $(Y_L \parallel Y_R)$ dan la cadena de salida Y.

   El proceso para hacer la descomposición se muestra en el pseudocódigo 3.8.

```
entrada:    bloque N_w de longitud n.
salida:     bloque Y_N
inicio
  para i = 0 hasta n − 1
    Y_N[i] = N_w mod s
    N_w = (N_w − Y_N[i])/s
fin
```

Pseudocódigo 3.8: Proceso de descomposición de L_w o R_w.

```
entrada:    la llave K,
            el tweak T,
            la cadena X de longitud b formada por el conjunto S
              de cardinalidad s,
            la función de cifrado F, y el número de rondas w.
salida:     La cadena cifrada Y.
inicio
  calcular T_R = T  mód 2^32 y T_L = (T − T_R)/2^32
  asignar l = r = b/2
  inicializar L_0 = Σ_{j=0}^{l−1} X[j] · s^j
  inicializar R_0 = Σ_{j=0}^{r−1} X[j + l] · s^j
  para i = 0 hasta i = w − 1
    si i es par
      L_{i+1} = L_i ⊞ F_K((T_R ⊕ i) · 2^{f−32} + R_i)     (mod s^l)
      R_{i+1} = R_i
    si i es impar
      R_{i+1} = R_i ⊞ F_K((T_L ⊕ i) · 2^{f−32} + L_i)     (mod s^r)
      L_{i+1} = L_i
  para i = 0 hasta i = l − 1
    Y_L[i] = L_w mod s
    L_w = (L_w − Y_L[i])/s
  para i = l hasta i = r − 1
    Y_R[i] = R_w mod s
    R_w = (R_w − Y_R[i])/s
  determinar Y = Y_L ‖ Y_R
fin
```

Pseudocódigo 3.9: Proceso de cifrado BC.

De forma general, el proceso de cifrado se describe en el pseudocódigo 3.9.

Proceso de descifrado BC^{-1}.

Para poder descifrar la cadena Y:

1. Se tiene que dividir en 2 la cadena Y, para obtener las subcadenas Y_L y Y_R con una longitud l y r respectivamente, de igual forma que se hizo con la cadena X en el proceso de cifrado.

2. Partiendo de que el proceso de descifrado se compone de w rondas, se define L_i y R_i y se inicializan en:

$$L_w = \sum_{j=0}^{l-1} Y_L[j] \cdot s^j \tag{3.15}$$

$$R_w = \sum_{j=0}^{r-1} Y_R[j] \cdot s^j \tag{3.16}$$

3. Ahora, comenzando con $i = w - 1$, para cada ronda $i \geq 0$.

   Si i es par:

$$L_i = L_{i+1} \boxminus E_K((T_R \oplus i) \cdot 2^{f-32} + R_{i+1}) \qquad (mod\ s^l) \tag{3.17}$$

$$R_i = R_{i+1} \tag{3.18}$$

   Si i es impar:

$$R_i = R_{i+1} \boxminus E_K((T_L \oplus i) \cdot 2^{f-32} + L_{i+1}) \qquad (mod\ s^r) \tag{3.19}$$

$$L_i = L_{i+1} \tag{3.20}$$

4. Finalmente se tienen que descomponer L_0 y R_0 (con el mismo proceso de descomposición descrito en el cifrado) para obtener a X_L y X_R, las cuales concatenadas ($X_L \parallel X_R$) dan la cadena de salida X.

   De forma general, el proceso de descifrado se describe en el pseudocódigo 3.10.

El modo de operación En cuanto al modo de operación de *BPS*, se puede decir que es un equivalente al modo de operación CBC (véase 2.2.10.2), ya que el bloque BC_n utiliza el texto cifrado de la salida del bloque BC_{n-1}, con la distinción de que en lugar de aplicar operaciones *xor* usa sumas modulares carácter por carácter, y de que no utiliza un VECTOR DE INICIALIZACIÓN, a pesar de soportar su uso.

Algo importante a resaltar de este modo de operación es que, en caso de que el texto en claro no tenga una longitud total que sea múltiplo de la longitud de bloque b, al cifrar el último bloque se recorre

entrada: la llave K,
el tweak T,
la cadena Y de longitud b formada por el conjunto S
de cardinalidad s,
la función de cifrado F,
el número de rondas w.
salida: La cadena X.
inicio
calcular $T_R = T \mod 2^{32}$ **y** $T_L = (T - T_R)/2^{32}$
asignar $l = r = b/2$
inicializar $L_w = \sum_{j=0}^{l-1} Y[j] \cdot s^j$
inicializar $R_w = \sum_{j=0}^{r-1} Y[j+l] \cdot s^j$
para $i = w-1$ hasta $i = 0$
si i es par:
$L_i = L_{i+1} \boxminus F_K((T_R \oplus i) \cdot 2^{f-32} + R_{i+1}) \quad (mod\ s^l)$
$R_i = R_{i+1}$
si i es impar:
$R_i = R_{i+1} \boxminus F_K((T_L \oplus i) \cdot 2^{f-32} + L_{i+1}) \quad (mod\ s^r)$
$L_i = L_{i+1}$
para $i = 0$ hasta $i = l-1$
$X_L[i] = L_w \ mod\ s$
$L_w = (L_w - X_L[i])/s$
para $i = l$ hasta $i = r-1$
$X_R[i] = R_w \ mod\ s$
$R_w = (R_w - X_R[i])/s$
determinar $X = X_L \parallel X_R$
fin

Pseudocódigo 3.10: Proceso de descifrado BC^{-1}.

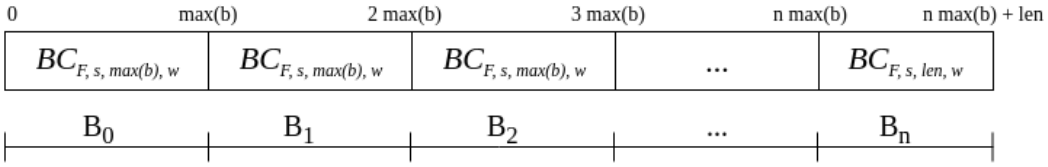


Figura 3.6: Corrimiento de cursor para la selección del último bloque en el modo de operación de *BPS*.

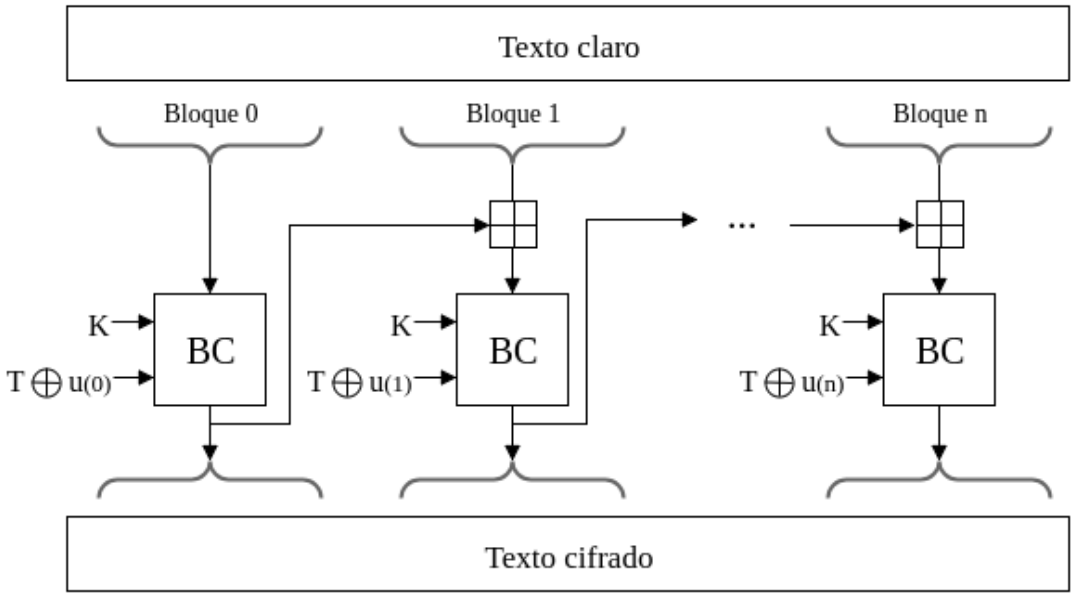


Figura 3.7: Modo de operación de *BPS*.

el cursor que determina el inicio del mismo, hasta que su longitud concuerde con b, esto se puede ver de forma gráfica en la figura 3.6.

En la figura 3.7 se ve de manera gráfica el funcionamiento del modo de operación de *BPS*.

Otra particularidad del modo de operación es el uso del contador u de 16 bits, que es utilizado para aplicar una operación *xor* al *tweak* T que entra a cada uno de los bloques BC. Recordando que T es de 64 bits, el *xor* se aplica a los 16 bits más significativos de ambas mitades de *tweak*, esto debido a cada mitad de *tweak* funciona de manera independiente en el cifrador BC, y a que no se desea un traslape entre el contador externo u y el contador i interno en BC.

Características generales Como se observó, *BPS* está basado en las redes Feistel, lo cual puede verse como una ventaja, debido al amplio estudio que tienen. Además, usa algoritmos de cifrado o funciones hash estandarizadas de forma interna, lo cual hace más comprensible y fácil su implementación.

BPS es un cifrado que preserva el formato capaz de cifrar cadenas de un longitud de 2 hasta $max(b) \cdot 2^b$ caracteres (donde $max(b)$ es el tamaño máximo de bloque), formadas por cualquier conjunto.

Se puede considerar que *BPS* es eficiente, debido a que la llave K usada en cada bloque BC es constante, y a que además usa un número reducido de operaciones internas en comparación con otros algoritmos de cifrado que preservan el formato.

Por último, se puede resaltar que el uso de *tweaks* protege a *BPS* de ataques de diccionario, los cuales

```
1  entrada: PAN p; información asociada d; llave k
2  salida:  token
3  inicio
4    S1 ← buscarPAN(p)
5    S2 ← buscarInfoAsociada(d)
6    si S1 y S2 = 0:
7      t ← RN(k)
8      insertar(t, p, d)
9    sino:
10     t ← S1
11   fin
12   regresar t
13 fin
```

Pseudocódigo 3.11: *TKR2*, método de tokenización

son fáciles de cometer cuando el dominio de la cadena a cifrar es muy pequeño.

Recomendaciones Se recomienda que el número de rondas w de la red Feistel sea 8, dado que es una número de rondas eficiente, y se ha estudiado la seguridad de *BPS* con este w.

Es recomendable que como *tweak* se use la salida truncada de una función hash, en donde la entrada de la función puede ser cualquier información relacionada a los datos que se deseen proteger, como por ejemplo fechas, lugares, o parte de los datos que no se deseen cifrar.

3.1.4.3. Algoritmo TKR2

En [27] se analiza formalmente el problema de la generación de TOKENS y se propone un algoritmo que no está basado en FPE. Hasta antes de la publicación de este documento, los únicos métodos para generar TOKENS cuya seguridad estaba formalmente demostrada eran los basados en FPE.

El algoritmo propuesto usa PRIMITIVAS CRIPTOGRÁFICAS para generar TOKENS aleatorios y almacena en una base de datos (CDV) la relación original de estos con los PAN. Según la clasificación del PCI se trata de una algoritmo tokenizador reversible *no criptográfico*; sin embargo tratarlo de esta forma (como *no criptográfico*) resulta un tanto confuso, dado que toda la contrucción del algoritmo y las pruebas de su seguridad se basan en fundamentos de la criptografía.

En el pseudocódigo 3.11 se muestra el proceso de tokenización, mientras que en 3.12 está la detokenización.

La mayor parte del proceso de tokenización y toda la detokenización son bastante fáciles de compren-

```
1  entrada: token t; información asociada d; llave k
2  salida: PAN
3  inicio
4    S1 ← buscarToken(t)
5    S2 ← buscarInfoAsociada(d)
6    si S1 y S2 = 0:
7      regresar error
8    sino:
9      p ← S1
10   fin
11   regresar p
12 fin
```

Pseudocódigo 3.12: *TKR2*, método de detokenización

der; lo único que queda por esclarecer es la función generadora de TOKENS aleatorios RN_k. Idealmente, esta función debe regresar un elemento uniformemente aleatorio del espacio de TOKENS. La propuesta que se hace en [27] para instanciar esta función se presenta en el pseudocódigo 3.13. Aquí, la variable *contador* mantiene un estado del algoritmo (mantiene su valor a lo largo de las distintas llamadas); el espacio de TOKENS contiene cadenas de longitud fija μ de un alfabeto AL cuya cardinalidad es l; el número de bits necesarios para enumerar a todo el alfabeto se guardan en $\lambda = \lceil \log_2 l \rceil$.

En el pseudocódigo 3.13 lo primero que se hace es utilizar una PRIMITIVA CRIPTOGRÁFICA f para generar una cadena binaria (hablaremos sobre este punto más adelante); esta cadena (nombrada X) se parte en subcadenas de λ bits; después se itera de manera consecutiva sobre estas subcadenas binarias, si la representación entera de la i-ésima está en el rango del alfabeto de los TOKENS, entonces se concatena al token resultado, sino, se pasa a la siguiente subcadena. La longitud de la cadena regresada por f debe ser, aproximadamente, $3\mu\lambda$: dado que se espera que el comportamiento de f sea EQUIPROBABLE, entonces el ciclo correrá un promedio de 2μ veces.

Existen varios candidatos viables para f: un cifrado de flujo (sección 2.3), pues el flujo de llave de estos produce cadenas de aspecto aleatorio; un cifrado por bloques (sección 2.2), utilizando un modo de operación de contador (sección 2.2.10.4); un TRUE RANDOM NUMBER GENERATOR (TRNG) para obtener secuencias de bits verdaderamente aleatorias.

Por último hay que aclarar que el algoritmo presentado en el pseudocódigo 3.11 debe recibir un par de modificaciones más: al momento de generar un TOKEN debe existir una validación que verifique que este sea único (para evitar que dos PAN tengan un mismo TOKEN); la base de datos debe estar cifrada, por lo que, antes de hacer inserciones y después de hacer consultas, deben existir las operaciones correspondientes.

```
1  entrada: llave k
2  salida:  token
3  inicio
4    X ← f(k, contador)
5    X_1, X_2, ..., X_m ← cortar(X, λ)
6    t ← ""
7    i ← 0
8    mientras |t| ≠ μ:
9      si entero(X_i) ≤ l:
10       t ← t + entero(X_i)
11     fin
12     i ← i + 1
13   fin
14   contador ← contador + 1
15   regresar t
16 fin
```

Pseudocódigo 3.13: *TKR2*, generación de TOKENS aleatorios

3.1.4.4. Algoritmo Hibrido Reversible

Longo, Aragona y Sala [39], propusieron en 2017, un algoritmo de tipo híbrido reversible. Este está basado en un cifrado de bloques con una llave secreta y una entrada adicional.

Se sabe que el número de una tarjeta (PAN) está conformado por tres partes concatenadas: el número que identifica al emisor de la tarjeta, el que identifica la cuenta y un número de verificación. En este algoritmo se reemplaza la primera parte con un TOKEN BANK IDENTIFIER NUMBER (BIN) y se cifra solo la parte que identifica la cuenta. Al final, se calcula un nuevo dígito de verificación.

Las entradas del algoritmo son la parte del PAN a cifrar y una entrada adicional. Esta última actúa como un *tweak* (véase sección 2.6), pues permite que se generen varios TOKENS para el mismo PAN.

El algoritmo necesita una función f pública que, dada una cadena de longitud m regrese una de longitud n (véase sección de funciones hash 2.4). Se toman solo cifrados cuyo tamaño de bloque sea de mínimo 128 bits. La función f se encarga de poner el relleno en la entrada para completar el bloque del cifrado y permitir la creación de varios TOKENS para el mismo PAN utilizando la misma llave en el proceso de cifrado. Finalmente, el algoritmo necesita una base de datos segura que se encargará de contener los pares PAN-TOKEN. Al momento de crear los TOKENS, se necesita acceder a la base de datos mediante una FUNCIÓN BOOLEANA *comprobar* que revisa si el TOKEN generado ya está almacenado en la base.

Como se desea obtener un TOKEN que tenga el mismo número de dígitos que el PAN (longitud l) ingresado, se deben tomar en cuenta solo una fracción de las posibles salidas del cifrado E; para resolver

```
entrada: PAN p; entrada_adicional u; llave k
salida:  token
inicio
  t = f(u,p)||[p̄]_b^s  (paso 1)
  c = E(k,t)  (paso 2)
  si  (c̄ mód 2^n) ≥ 10^l
    t = c
    Regresar al paso 2.
  fin
  token = [c̄ mód 2^n]_10^l
  si comprobar(token) = verdadero
    u = u + 1
    Regresar al paso 1.
  fin
  regresar token
fin
```

Pseudocódigo 3.14: Híbrido reversible, método de tokenización

este problema, se utiliza un método conocido como el CIFRADO DE CAMINATA CÍCLICA.

Notación A continuación se definen una serie de notaciones que se utilizarán en el algoritmo:

- M Tamaño de bloque del cifrador por bloques que se usará.
- l Longitud de la entrada. En este caso, $13 \geq l \geq 19$.
- n Número de bits necesarios para representar a la entrada: $n = log_2(10^l)$.
- $[y]_b^s$ Indica que y es menor que b^s: $y < b^s$.
- $\bar{x}$ Representación de x en una cadena binaria cuando x es representado en su forma decimal y viceversa.

El algoritmo de tokenización es el siguiente:

3.1.4.5. Tokenización mediante generador de números pseudoaleatorios

En la sección 3.1.2.3 se habló sobre el estándar del NIST para la generación de números aleatorios. En esta mostramos cómo estos métodos pueden ser utilizadas para tokenizar y detokenizar números de tarjetas. Primero se muestran dos posibles instanciaciones con PRIMITIVAS CRIPTOGRÁFICAS del método general

```
1  entrada: número n de bytes deseados.
2  salida:  arreglo de n bytes.
3  inicio
4    longitud ← tamanioDeHash()
5    numeroDeBloques ← ⌈longitud/n⌉
6    datos ← semilla
7    para_todo i en numeroDeBloques:
8      resultado ||= hash(datos)
9      datos += 1
10   fin
11   regresar n bytes de resultado
12 fin
```

Pseudocódigo 3.15: Generación de bits pseudoaleatorios mediante función hash

descrito en 3.1.2.3: una basada en funciones hash (sección 2.4) y otra basada en cifradores por bloque (sección 2.2); ambas presentadas en [33].

DRBG basado en funciones hash El método genérico define un solo valor crítico (un valor que se debe mantener en secreto, pues la seguridad en generador depende de esto): la semilla, tanto al inicio, como su valor a lo largo de su vida útil. Este método introduce un segundo valor crítico: una constante C cuyo valor depende también de la entrada de entropía. Por tanto, las funciones de instanciación, cambio de semilla y desinstanciación se deben modificar para incluir a este segundo valor.

En el pseudocódigo 3.15 se muestra la función de generación de bits pseudoaleatorios basada en una función hash. *tamanioDeHash* regresa el tamaño de los *digest* retornados por la función hash usada.

DRBG basado en cifrador por bloque Al igual que el generador basado en funciones hash, este método también introduce un segundo valor crítico: el valor de la llave utilizada por el cifrador por bloque subyacente. Esta llave se genera a partir de la entrada de entropía y debe ser tratada de igual forma que la semilla por las funciones de instanciación, cambio de semilla y desinstanciación.

En el pseudocódigo 3.16 se muestra la función de generación de bits psudoaleatorios basada en un cifrador por bloques.

Uso como algoritmo tokenizador Un DRBG puede ser utilizado de manera bastante similar a la función RN del algoritmo TKR (sección 3.1.4.3): en lugar de utlizar la función f como método para obtener bits pseudoaleatorios, se ocupa el generador pseudoaleatorio. El pseudocódigo para esto se muestra en 3.17; la función *drbg* es una llamada a la operación de generación de bits pseudoaleatorios de cualquiera

```
entrada: número n de bytes deseados.
salida:  arreglo de n bytes.
inicio
  mientras longitud(resultado) ≤ n:
    semilla += 1
    resultado ||= cifrar(semilla)
  fin
  regresar n bytes de resultado
fin
```

Pseudocódigo 3.16: Generación de bits pseudoaleatorios mediante cifrador por bloques

de los dos generadores presentados anteriormente.

Al igual que con TKR, ambas operaciones (tokenización y detokenización) deben tener el soporte de una base de datos: al generar un nuevo TOKEN este se guarda en la base de datos, y la operación de detokenización es solamente una consulta en la base.

3.1.5. Diseño de programa tokenizador

En la sección anterior se enlistan los algoritmos tokenizadores que implementaremos. En esta se argumentan algunas de las decisiones tomadas con respecto a las tecnologías usadas y se usa el UNIFIED MODELING LANGUAGE (UML) para describir los aspectos más significativos del programa.

3.1.5.1. Vista estática del programa

En el diagrama de la figura 3.8 se muestra una vista estática general del programa. Las clases se dividen en dos paquetes: implementaciones y utilidades. Las utilidades abarcan estructuras de datos, interfaces, clases de excepciones y clases de prueba; todas ellas no tienen que ver de forma directa con criptografía, sino que se trata de código de soporte para las implementaciones. En las implementaciones van no solamente los algoritmos presentados en la sección 3.1.4, sino que todas las demás clases (generalizaciones, utilidades, implementaciones de soporte) que están relacionadas con los algoritmos tokenizadores; en el caso de este paquete, sí todo está relacionado o con criptografía o con un entorno bancario.

Retornando al diagrama de 3.8: muestra solamente las clases e interfaces directamente relacionadas con los algoritmos tokenizadores; en las próximas secciones se mostrarán algunas otras vistas. Las utilidades contienen las interfaces para definir los distintos tipos de funciones y la estructura de datos (Arreglo) que se usa de manera constante en todo el programa. Las implementaciones muestran las interfaces que definen y clasifican a los algoritmos tokenizadores; también se muestra (por razones de espacio, solo el nombre) las clases concretas de los sistemas tokenizadores.

```
1  entrada: llave k
2  salida:  token
3  inicio
4    X ← drbg()
5    X_1, X_2, ..., X_m ← cortar(X, λ)
6    t ← ""
7    i ← 0
8    mientras |t| ≠ μ:
9      si entero(X_i) ≤ l:
10       t ← t + entero(X_i)
11     fin
12     i ← i + 1
13   fin
14   contador ← contador + 1
15   regresar t
16 fin
```

Pseudocódigo 3.17: Generación de *tokens* mediante DRBG

Las interfaces de las funciones utilizan plantillas para definir las propias firmas de los métodos abstractos que declaran. Esta es una forma bastante efectiva para hacer diseños débilmente acoplados (ver ACOPLAMIENTO): por ejemplo, las redes Fesitel necesitan de una función de ronda sobre la que *no necesitan* saber nada en particular, solo necesitan saber que la clase define un método **operar**, que es con lo que funciona el algoritmo definido por la red.

El Arreglo es una implementacion propia de una estuctura de datos de almacenamiento secuencial. Es solamente una interfaz a una sección de memoria (un arreglo tradicional); sin embargo, lo importante de este es que imita el formato de las estructuras de la librería estándar de C++ (esquema de constructores y destructores para gestión de memoria, uso de plantillas para programación genérica, etcétera). El arreglo de dígitos es una especialización que mantiene una representación interna tanto en cadena como en número; este arreglo es el que ocupan para comunicarse todos los algoritmos tokenizadores. Por esta última razón es por la que se mantienen las dos representaciones internas: algunos métodos requieren interpretar las entradas como números y otros como cadenas.

Los algoritmos tokenizadores implementan la interfaz definida por las funciones con inverso definiendo el tipo de ida y de vuelta como un arreglo de dígitos (PAN y TOKEN). Las interfaces intermedias (algoritmos reversibles e irreversibles) permiten definir un comportamiento genérico para cada tipo de algoritmo; estas a su vez declaran los métodos abstractos que los algoritmos concretos deben de implementar.

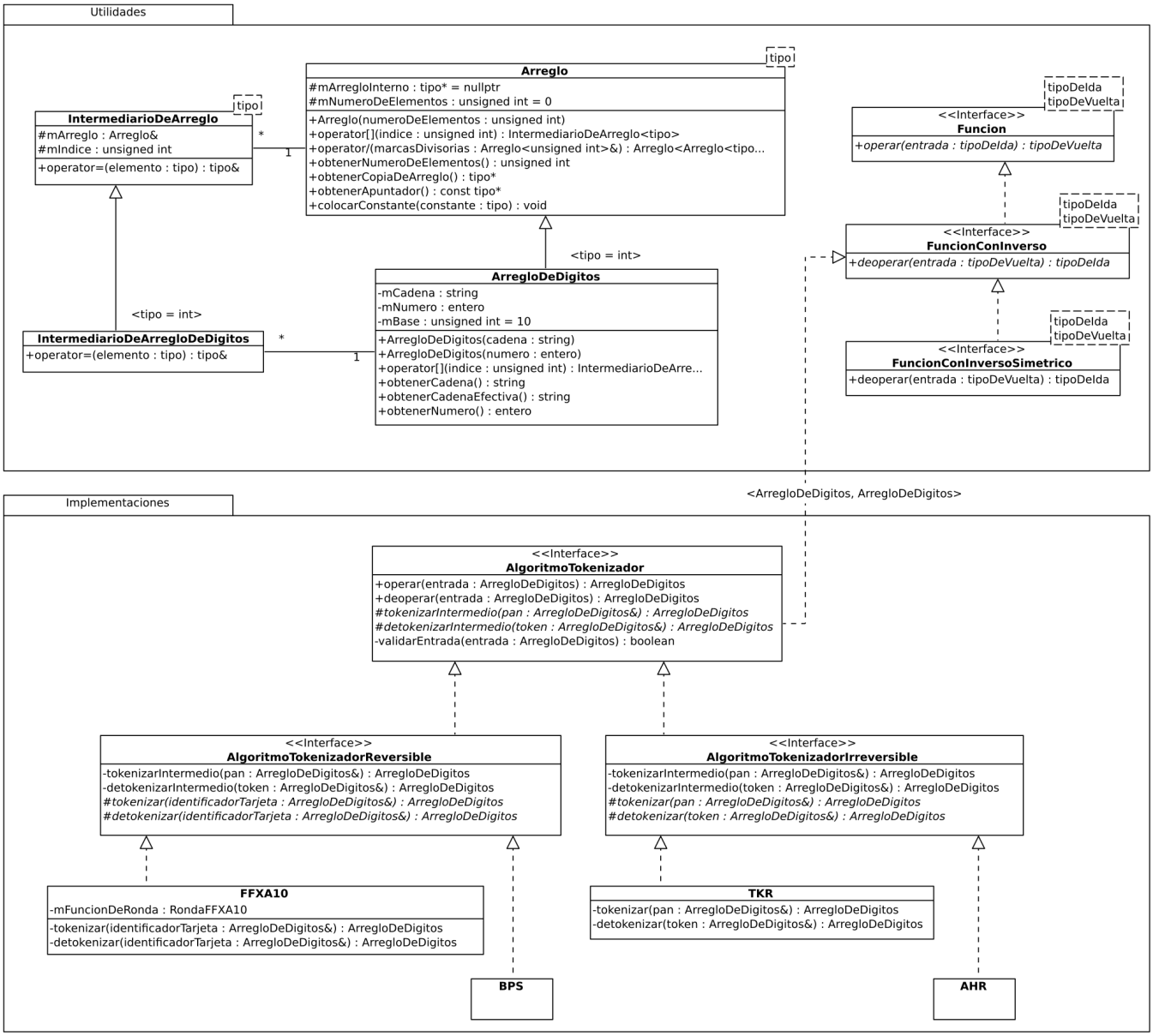


Figura 3.8: Diagrama de clases general.

Clases de FFX En la figura 3.9 se muestran las clases que conforman al módulo de FFX. Sin contar a las tres interfaces de funciones (que forman parte del paquete de utilidades) todas las clases son del paquete de implementaciones.

La clase de la red Feistel implementa la interfaz de una función con inverso (tiene tanto operación de ida, como de vuelta) y se compone (por medio de una relación de composición) de una función, utilizada como fucnión de ronda, y de una función con inverso, utilizada como operador de combinación. Ambas clases hijas (redes Feistel alternantes o desbalanceadas) contienen un indicador de desbalanceo y sobreescriben ambas operaciones de la superclase; la red feistel alternante (ver sección 2.2.3) agrega una nueva función de ronda: una para las pares y otra para las impares.

En la parte superior de diagrama se muestran las dos posibles operaciones de combinación que soporta ffx: una a nivel de bloque y la otra nivel de caracter. La única clase en donde estas dos opciones son reestrictivas es desde la construcción de FFXA10; el desacoplamiento dado por las interfaces permite que lo único que necesite saber la red es que tiene una función que recibe un arreglo y entrega un arreglo.

Otra clase con la misma esructura que las dos anteriores es la de la función de ronda de FFXA10: implementa la interfaz de una función con inverso simétrico (que a su vez implementa un función con inverso); una instancia de esta clase en FFX es usada para construir a la red Feistel con la que se opera.

La clase de FFX es solamente un medio de comunicación con la red Feistel interna, esto es, no agrega ninguna lógica extra a los procesos de operación y operación inversa. Lo mismo ocurre con FFXA10; solo que mientras el contructor de la superclase es abierto a personalizar la red (FFX debería servir para cualquier cifrado que preserva el formato, no solo para un alfabeto de dígitos), el contructor de FFXA10 tiene parámetros bastante limitados, y su construcción de la superclase y la red Feistel ya es predefinida según la descripción de esta colección hecha en la sección 3.1.4.1. Es esta última clase (FFXA10) la que implementa la interfaz de los algoritmos tokenizadores reversibles.

Clases de TKR En la figura 3.10 se muestran las clases relacionadas al módulo de TKR. Al igual que con el diagrama anterior, la gran mayoría de las clases pertenecen al paquete de implementaciones; las excepciones marcan el paquete al que pertenecen.

Este es el primer diagrama en donde es necesario mostrar el esquema de acceso a una fuente de datos. Este es descrito por una interfaz llamada CDV (Card Data Vault), que define las operaciones que debe de proveer cualquier fuente de datos que se desee usar dentro del programa. En un primer esquema existen dos clases concretas que implementan esta interfaz: un acceso trivial y un acceso a MySQL (ver sección 3.1.5.2). El acceso trivial permite definir métodos «de prueba» cuando se introduce una nueva operación en la interfaz; de esta forma, el código cliente puede ocupar esta implementación en lo que se desarrolla la verdadera, con conexión a base de datos. Esta manera de separar interfaz de implementación para el

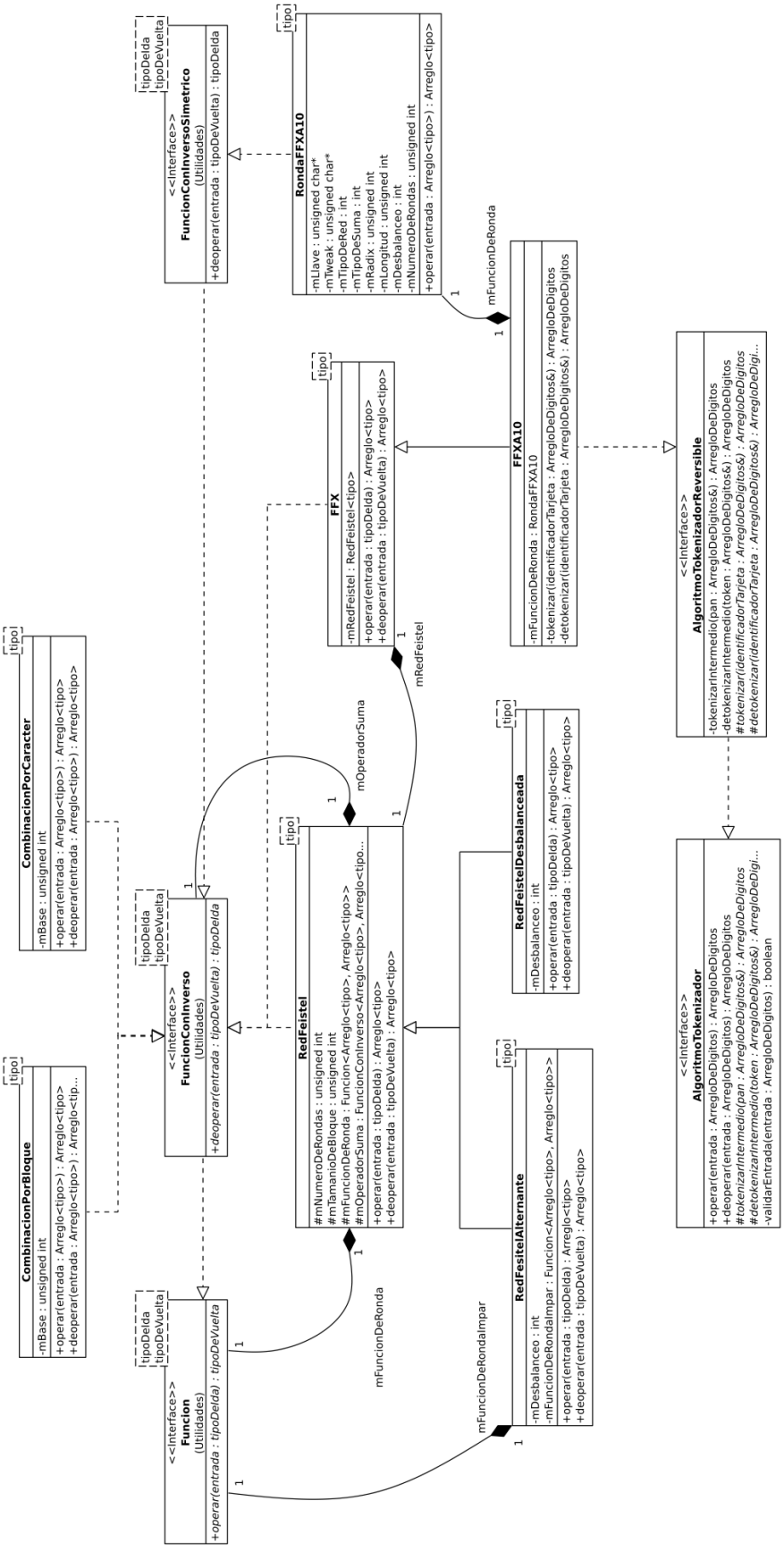


Figura 3.9: Diagrama de clases de módulo de FFX.

acceso a datos es un patrón de arquitectura conocido como Data Access Object (DAO); a grandes rasgos permite separar la definición de *qué* datos necesita la aplicación, del *cómo* los obtendrá; de esta manera, un cambio en el gestor de base de datos no debe afectar al código que usa la interfaz.

En la parte inferior del diagrama se muestra la clase *Registro*. Esta representa la única clase de datos del entorno del modelo; contiene la relación entre un PAN y un token. Como aclaración, es la única clase de datos del modelo del módulo del programa tokenizador; la interfaz web introducirá muchas otras clases al modelo.

TKR, como se explicó en la sección 3.1.4.3, necesita de una función que genere tokens pseudoaleatorios (la función RN) la cuál necesita de una función proveedora de bits pseudoaleatorios. En el diagrama se muestran tres versiones de esta última función: una trivial, una basada en AES y la tercera basada en un DRBG (costado izquierdo del diagrama). La trivial es una primera implementación que genera números pseudoaleatorios de un modo que no es criptográficamente seguro. La basada en AES es tal cuál se decribe en el artículo [27]. Por último, la basada en un DRBG, es la que permite definir el algoritmo tokenizador que se describe en la sección 3.1.4.5; para este algoritmo se ocupa la misma estructura que para TKR, sin embargo, lo que importa en esta implementación es la propia creación del generador, que se muestra en secciones posteriores.

La relación entre la función RN y su mecanismo de generación interno está totalmente desacoplada: lo único que necesita la clase de la función RN es otra función que reciba enteros sin signo y entregue arreglos de bytes. Este modelo de desacoplamiento no se imitó en la relación entre la clase de TKR y la función RN porque la especificación de TKR es muy específica respecto a qué función usar.

La clase de TKR implementa la interfaz de un algoritmo tokenizador irreversible, por lo que debe (programación por contrato) implementar los métodos para tokenizar y detokenizar.

Clases de DRBG Las clases relacionadas con el generador de números pseudoaleatorio se muestran en la figura 3.11. Nuevamente, todas las clases pertenecen al paquete de las implementaciones.

Un DRBG implementa la interfaz de una función que recibe enteros sin signo y entrega arreglos de bytes. Esta clase define la estructura general de un generador: las cinco funciones definidas por el estámdar del NIST (ver sección **??**) y el conjunto de valores miembro que conforman el estado del generador. El DRBG declara la función abstracta para generar bytes; todos los posibles generadores concretos deben implementar esta función para poder tener un generador completo.

La clase del generador necesita una fuente de entropía (o fuente de aleatoriedad). Siguiendo el esquema de débil acomplamiento dado por las interfaces de las funciones, la fuente de entropia es una función que recibe un entero y regresa un arreglo de bytes. En el costado izquierdo del diagrama se muestra una de las posibles fuentes de entropia (aleatoriedad trivial), la cual lee de un archivo la entropía.

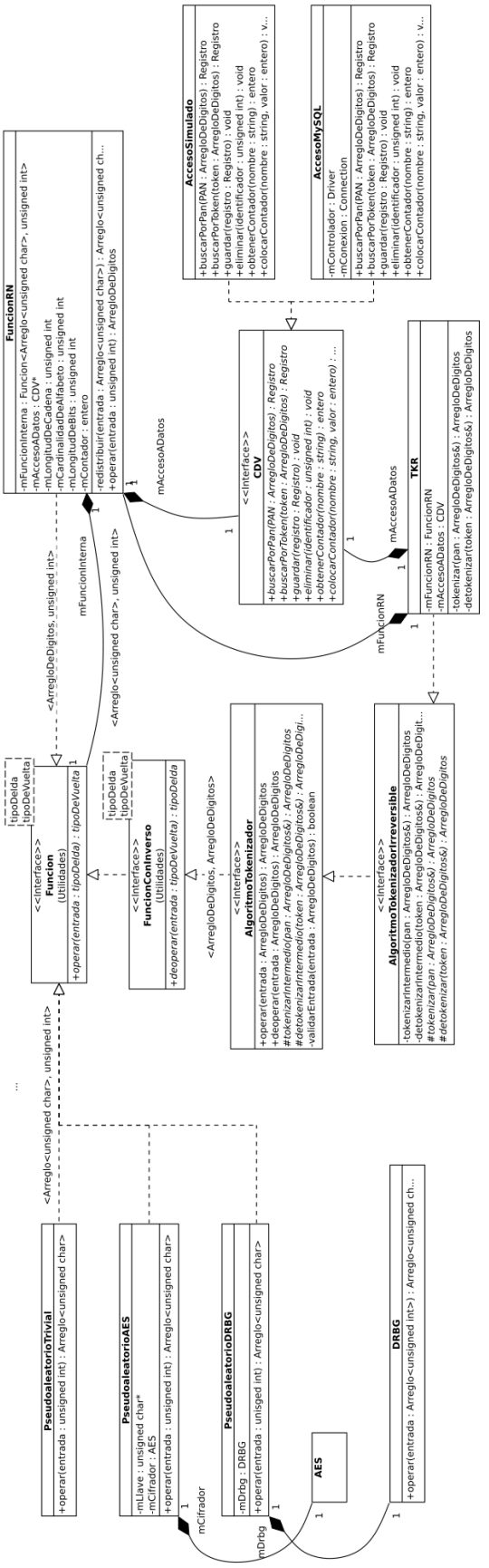


Figura 3.10: Diagrama de clases de módulo de TKR.

Existen dos implementaciones concretas para el generador de números pseudoaleatorios: una basada en una función hash y la otra basada en un cifrador por bloques (sección **??**). Ambos agregan sus propios datos miembro al estado del generador, sobrecargan las funciones de cambio de semilla y desintanciación e implementan el método abstracto de la superclase. También se utilizan enumeraciones para indicar al código cliente cuales son las distintas funciones hash y cifradores por bloque que pueden ser ocupados.

3.1.5.2. Selección de tecnologías

Las primeras decisiones tomadas con respecto a la implementación y al diseño fueron sobre el paradigma de programación y el lenguaje a utilizar. Para tomar estas decisiones se tomaron en cuenta varias consideraciones: primero, el paradigma orientado a objetos ofrece considerables ventajas con respecto a la programación estructurada, por lo que, dentro de lo posible, se buscaría un lenguaje con soporte a este paradigma; segudo, las implementaciones criptográficas tienen altos requerimientos de rendimiento, por lo que, de los posibles lenguajes, necesitabamos elegir uno que, si bien no el más rápido, sí se encontrara entre los de mayor velocidad.

A lo largo de nuestra carrera hemos tenido contacto con bastantes lenguajes que, si bien no dominamos, sí poseemos una base firme como para ser usada: C, C++, Java, Python, Javascript y PHP. Por la cuestión del rendimiento, todos los lenguajes interpretados (Python, Javascript, PHP) quedaron fuera de consideración; la decisión (dentro de los lenguajes con soporte al paradigma orientado a objetos) quedó entre Java y C++: Java (aún en las últimas versiones, en donde se han hecho considerables progresos) es mucho más lento que C++, dado que el trabajo de la máquina virtual en tiempo de ejecución es bastante considerable; por lo tanto, si de un lenguaje orientado a objetos se trataba, sería C++. La última decisión se dió entre C y C++: orientación a objetos contra rendimiento. Al final se optó por la orientación a objetos: aún cuando C es más rápido que C++, la diferencia no es tan grande, mientras que un programa con un buen diseño orientado a objetos sí puede ser mucho más mantenible que uno con un enfoque estructurado.

La verisón de C++ que utilizamos es C++14. Por razones de compatibilidad con algunas de nuestras dependencias (en particular, el conector de la base de datos) no pudimos utilizar la última versión al momento, C++17.

El gestor de base de datos que más hemos utilizado en la carrera es MySQL, por lo que es el que seleccionamos para el programa tokenizador; en realidad, se trata de una bifurcación de MySQL: MariaDB, cuyo uso ha sido ampliamente difundido en varias distribuciones de Linux.

Otra decisión importante a tomar antes de hacer las implementaciones de los algoritmos tokenizadores fue la librería de funciones criptográficas que utilizaríamos. De todas las posibles librerías que están validadas por el NIST y que cuentan con una licencia para el uso público, al final hicimos pruebas con dos: Openssl y Crypto++. Openssl está escrita en C y cuenta con un historial que le da mucha reputación (es la librería que utiliza Opengpg y Openssh). Cryptopp cuenta con una extensa red de colaboradores,

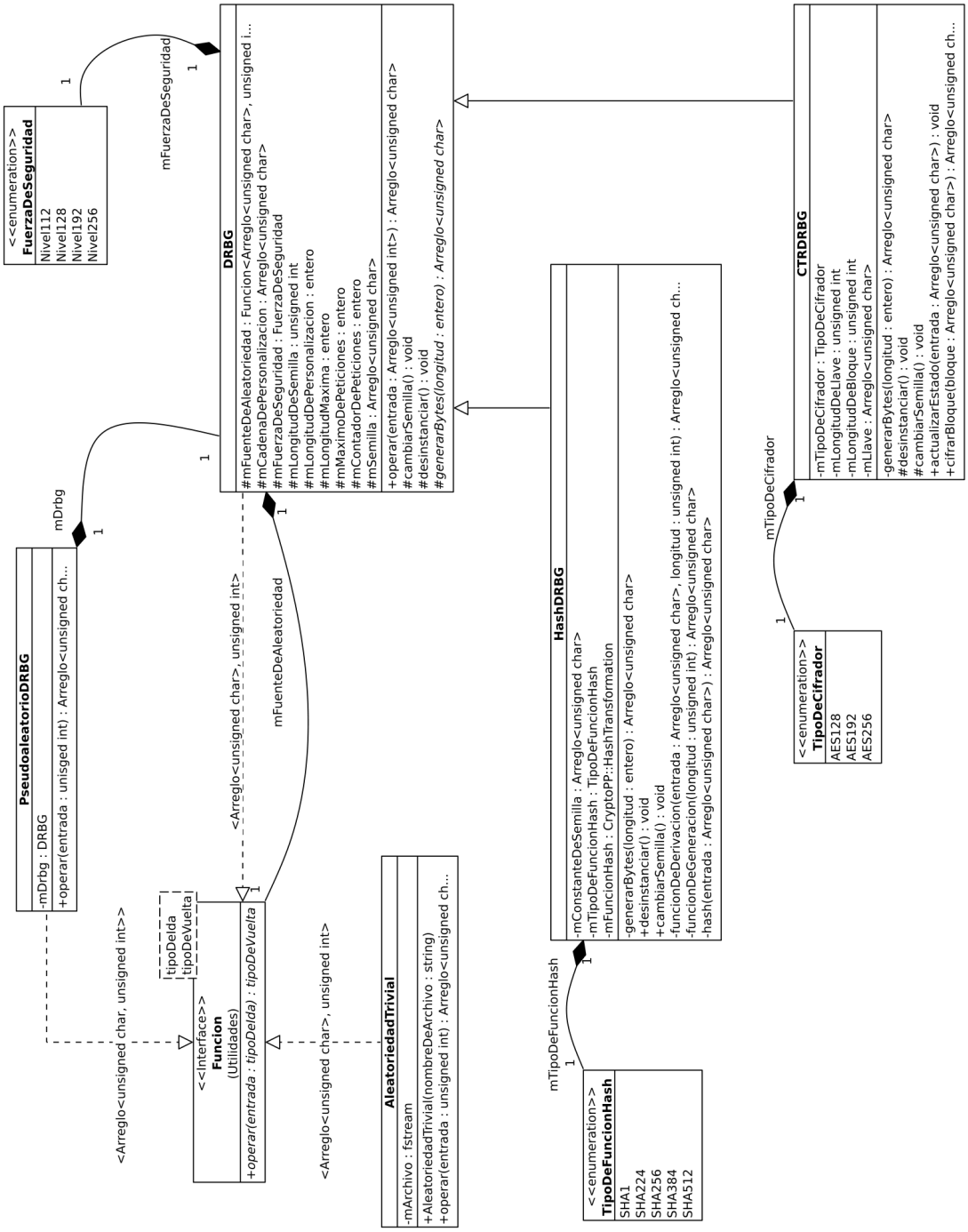


Figura 3.11: Diagrama de clases de módulo de DRBG.

implementa una gran cantidad de algoritmos y está escrita sólo en C++. De ambas, la que encontramos más fácil de usar fue Crypto++, por lo que es la que utilizamos en nuestras implementaciones.

Bibliografía

[1] Alfred Menezes, Paul C. van Oorschot y Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 0-8493-8523-7 (vid. págs. 4, 9, 22, 29, 34, 38, 119, 123).

[2] Hans Delfs y Helmut Knebl. *Introduction to Cryptography - Principles and Applications*. Information Security and Cryptography. Springer, 2007. ISBN: 978-3-540-49243-6. DOI: 10.1007/3-540-49244-5. URL: HTTPS://DOI.ORG/10.1007/3-540-49244-5 (vid. págs. 4, 9, 34, 38).

[3] Claude E. Shannon. "Communication theory - Exposition of fundamentals". En: *Trans. of the IRE Professional Group on Information Theory (TIT)* 1 (1953), págs. 44-47. DOI: 10.1109/TIT.1953.1188568. URL: HTTPS://DOI.ORG/10.1109/TIT.1953.1188568 (vid. pág. 5).

[4] Whitfield Diffie y Martin E. Hellman. "New directions in cryptography". En: *IEEE Trans. Information Theory* 22.6 (1976), págs. 644-654. DOI: 10.1109/TIT.1976.1055638. URL: HTTPS://DOI.ORG/10.1109/TIT.1976.1055638 (vid. pág. 7).

[5] Ronald L. Rivest, Adi Shamir y Leonard M. Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". En: *Commun. ACM* 21.2 (1978), págs. 120-126. DOI: 10.1145/359340.359342. URL: HTTP://DOI.ACM.ORG/10.1145/359340.359342 (vid. pág. 7).

[6] Phillip Rogaway. *A Synopsis of Format-Preserving Encryption*. 2010. URL: HTTP://WEB.CS.UCDAVIS.EDU/~ROGAWAY/PAPERS/SYNOPSIS.PDF (vid. págs. 11, 46).

[7] Bruce Schneier y John Kelsey. "Unbalanced Feistel Networks and Block Cipher Design". En: *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*. Ed. por Dieter Gollmann. Vol. 1039. Lecture Notes in Computer Science. Springer, 1996, págs. 121-144. ISBN: 3-540-60865-6. DOI: 10.1007/3-540-60865-6_49. URL: HTTPS://DOI.ORG/10.1007/3-540-60865-6_49 (vid. pág. 11).

[8] Ross J. Anderson y Eli Biham. "Two Practical and Provably Secure Block Ciphers: BEARS and LION". En: *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*. Ed. por Dieter Gollmann. Vol. 1039. Lecture Notes in Computer Science. Springer, 1996, págs. 113-120. ISBN: 3-540-60865-6. DOI: 10.1007/3-540-60865-6_48. URL: HTTPS://DOI.ORG/10.1007/3-540-60865-6_48 (vid. pág. 12).

[9] Stefan Lucks. "Faster Luby-Rackoff Ciphers". En: *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*. Ed. por Dieter Gollmann. Vol. 1039. Lecture Notes in Computer Science. Springer, 1996, págs. 189-203. ISBN: 3-540-60865-6. DOI: 10.1007/3-540-60865-6_53. URL: HTTPS://DOI.ORG/10.1007/3-540-60865-6_53 (vid. pág. 12).

[10] Debrup Chakraborty y Francisco Rodríguez-Henríquez. "Block Cipher Modes of Operation from a Hardware Implementation Perspective". En: *Cryptographic Engineering*. Ed. por Çetin Kaya Koç. Springer, 2009, págs. 321-363. ISBN: 978-0-387-71816-3. DOI: 10.1007/978-0-387-71817-0_12. URL: HTTPS://DOI.ORG/10.1007/978-0-387-71817-0_12 (vid. pág. 22).

[11] William Stallings. *Cryptography and network security - principles and practice (6. ed.)* Pearson, 2014. ISBN: 978-0-13-335469-0 (vid. págs. 29, 119).

[12] Alan G. Konheim. *Computer Security and Cryptography*. Wiley, 2007. ISBN: 978-0-471-94783-7 (vid. pág. 29).

[13] Scott R. Fluhrer, Itsik Mantin y Adi Shamir. "Weaknesses in the Key Scheduling Algorithm of RC4". En: *Selected Areas in Cryptography, 8th Annual International Workshop, SAC 2001 Toronto, Ontario, Canada, August 16-17, 2001, Revised Papers*. Ed. por Serge Vaudenay y Amr M. Youssef. Vol. 2259. Lecture Notes in Computer Science. Springer, 2001, págs. 1-24. ISBN: 3-540-43066-0. DOI: 10.1007/3-540-45537-X_1. URL: HTTPS://DOI.ORG/10.1007/3-540-45537-X_1 (vid. pág. 32).

[14] Matthew J. B. Robshaw y Olivier Billet, eds. *New Stream Cipher Designs - The eSTREAM Finalists*. Vol. 4986. Lecture Notes in Computer Science. Springer, 2008. ISBN: 978-3-540-68350-6. DOI: 10.1007/978-3-540-68351-3. URL: HTTPS://DOI.ORG/10.1007/978-3-540-68351-3 (vid. pág. 32).

[15] Christophe De Cannière y Bart Preneel. "Trivium". En: *New Stream Cipher Designs - The eSTREAM Finalists*. Ed. por Matthew J. B. Robshaw y Olivier Billet. Vol. 4986. Lecture Notes in Computer Science. Springer, 2008, págs. 244-266. ISBN: 978-3-540-68350-6. DOI: 10.1007/978-3-540-68351-3_18. URL: HTTPS://DOI.ORG/10.1007/978-3-540-68351-3_18 (vid. pág. 32).

[16] Steve Babbage, Christophe De Cannière, Anne Canteaut y col. "The eSTREAM Portfolio (rev. 1)". En: (2008). URL: HTTP://WWW.ECRYPT.EU.ORG/STREAM/PORTFOLIO_REVISION1.PDF (vid. pág. 32).

[17] Alaa Hussein Al-Hamami y Ghossoon M. Waleed al-Saadoo. *Handbook of research on threat detection and countermeasures in network security*. 1.ª ed. IGI Global, 2015 (vid. pág. 34).

[18] Prakash C. Gupta. *Cryptography and Network Security*. PHI Learning, 2015 (vid. pág. 34).

[19] Dhiren R. Patel. *Information security - Theory and Practice*. Prentice-Hall of India, 2008 (vid. pág. 38).

[20] Cuauhtémoc Mancillas López. "Studies on Disk Encryption". Centro de Investigación y Estudios Avanzados del Instituto Politécnico Nacional, 2013 (vid. pág. 44).

[21] Moses Liskov, Ronald L. Rivest y David A. Wagner. "Tweakable Block Ciphers". En: *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*. Ed. por Moti Yung. Vol. 2442. Lecture Notes in Computer Science. Springer, 2002, págs. 31-46. ISBN: 3-540-44050-X. DOI: 10.1007/3-540-45708-9_3. URL: HTTPS://DOI.ORG/10.1007/3-540-45708-9_3 (vid. pág. 44).

[22] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 1969. ISBN: 0201038021. URL: HTTP://WWW.WORLDCAT.ORG/OCLC/310551264 (vid. págs. 47, 132).

[23] International Organization for Standarization. *ISO/IEC 7812*. 5.ª ed. 2017, pág. 7. URL: HTTPS://WWW.ISO.ORG/STANDARD/70484.HTML (vid. pág. 48).

[24] International Organization for Standardization. *ISO 9362*. 4.ª ed. 2014, pág. 6. URL: HTTPS://WWW.ISO.ORG/STANDARD/60390.HTML (vid. pág. 48).

[25] Abhay Bhargav. *PCI compliance: The Definitive Guide*. CRC Press, 2015 (vid. pág. 48).

[26] Payment Card Industry Security Standards Council. *Tokenization Product Security Guidelines – Irreversible and Reversible Tokens*. 2015. URL: HTTPS://WWW.PCISECURITYSTANDARDS.ORG/DOCUMENTS/TOKENIZATION_PRODUCT_SECURITY_GUIDELINES.PDF (vid. págs. 52, 57-60).

[27] Sandra Diaz-Santiago, Lil María Rodríguez-Henríquez y Debrup Chakraborty. "A cryptographic study of tokenization systems". En: *Int. J. Inf. Sec.* 15.4 (2016), págs. 413-432. DOI: 10.1007/S10207-015-0313-X. URL: HTTPS://DOI.ORG/10.1007/S10207-015-0313-X (vid. págs. 52, 100, 101, 110).

[28] Payment Card Industry Security Standards Council. *Data Security Standard - Version 3.2*. 2016. URL: HTTPS://WWW.PCISECURITYSTANDARDS.ORG/DOCUMENTS/PCI_DSS_V3-2.PDF (vid. págs. 52, 54).

[29] Information Technology Laboratory National Institute of Standards y Technology. *FIPS PUB 140-2 - Security Requirements for cryotographic modules*. 2001. URL: HTTPS://CSRC.NIST.GOV/CSRC/MEDIA/PUBLICATIONS/FIPS/140/2/FINAL/DOCUMENTS/FIPS1402.PDF (vid. págs. 53, 57, 59, 69).

[30] Payment Card Industry Security Standards Council. *Payment Application Data Security Standard - Requirements and Security Assessment Procedures - Version 3.0*. 2013. URL: HTTPS://WWW.PCISECURITYSTANDARDS.ORG/MINISITE/EN/DOCS/PA-DSS_V3.PDF (vid. págs. 54, 55).

[31] Elaine Barker. *NIST Special Publication 800-57 - Recommendation for Key Management*. 2016. URL: HTTP://DX.DOI.ORG/10.6028/NIST.SP.800-57PT1R4 (vid. págs. 55, 63, 66).

[32] Elaine Barker, Miles Smid, Dennis Branstad y col. *NIST Special Publication 800-130 - A Framework for Designing Cryptographic Key Management Systems*. 2013. URL: HTTP://DX.DOI.ORG/10.6028/NIST.SP.800-130 (vid. págs. 55, 63).

[33] Elaine Barker y John Kelsey. *NIST Special Publication 800-90A - Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. 2015. URL: HTTP://DX.DOI.ORG/10.6028/NIST.SP.800-90AR1 (vid. págs. 57, 58, 60, 63, 77, 104).

[34] Lily Chen. *NIST Special Publication 800-108 - Recommendation for Key Derivation Using Pseudorandom Functions*. 2009. URL: HTTP://NVLPUBS.NIST.GOV/NISTPUBS/LEGACY/SP/NISTSPECIALPUBLICATION800-108.PDF (vid. pág. 63).

[35] Elaine Barker y Allen Roginsky. *NIST Special Publication 800-133 - Recommendation for Cryptographic Key Generation*. 2012. URL: HTTP://DX.DOI.ORG/10.6028/NIST.SP.800-133 (vid. pág. 63).

[36] Mihir Bellare, Phillip Rogaway y Terence Spies. "The FFX Mode of Operation for Format-Preserving Encryption". Ver. 1.0. En: (2009) (vid. pág. 92).

[37] Mihir Bellare, Phillip Rogaway y Terence Spies. "The FFX Mode of Operation for Format-Preserving Encryption". Ver. 1.1. En: (2010) (vid. pág. 93).

[38] Eric Brier, Thomas Peyrin y Jacques Stern. "BPS: a Format-Preserving Encryption Proposal". En: (2010) (vid. pág. 94).

[39] Riccardo Aragona, Riccardo Longo y Massimiliano Sala. "Several proofs of security for a tokenization algorithm". En: *Appl. Algebra Eng. Commun. Comput.* 28.5 (2017), págs. 425-436. DOI: 10.1007/s00200-017-0313-3. URL: HTTPS://DOI.ORG/10.1007/S00200-017-0313-3 (vid. pág. 102).

[40] Grady Booch, Robert A. Maksimchuk, Michael W. Engle y col. *Object-oriented analysis and design with applications, Third Edition*. Addison Wesley object technology series. Addison-Wesley, 2007. ISBN: 978-0-201-89551-3 (vid. pág. 119).

[41] Donald Beaver, Silvio Micali y Phillip Rogaway. "The Round Complexity of Secure Protocols (Extended Abstract)". En: *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*. Ed. por Harriet Ortiz. ACM, 1990, págs. 503-513. ISBN: 0-89791-361-2. DOI: 10.1145/100216.100287. URL: HTTP://DOI.ACM.ORG/10.1145/100216.100287 (vid. págs. 120, 122).

[42] Payment Card Industry Security Standards Council. *Data Security Standard (DSS) and Payment Application Data Security Standard (PA-DSS) - Glossary of Terms, Abbreviations, and Acronyms - Version 1.2*. 2008. URL: HTTPS://WWW.PCISECURITYSTANDARDS.ORG/PDFS/PCI_DSS_GLOSSARY.PDF (vid. pág. 120).

[43] Dieter Gollmann, ed. *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*. Vol. 1039. Lecture Notes in Computer Science. Springer, 1996. ISBN: 3-540-60865-6. DOI: 10.1007/3-540-60865-6. URL: HTTPS://DOI.ORG/10.1007/3-540-60865-6.

Glosario

Glosario de términos criptográficos y matemáticos. Las principales fuentes bibliográficas usadas son [1] y [11]; en caso de tratarse de una fuente distinta, se indica en la entrada en particular.

1. Acoplamiento

(*Coupling*) Medida de la fuerza de asociación establecida por la conexión de dos módulos dentro de un sistema. Un acoplamiento fuerte complica a un sistema, dado que es más difícil de entender, cambiar o corregir. La complejidad de un programa se puede reducir buscando el menor acoplamiento posible entre módulos [40]. 1, 106

2. Autenticación de origen

Tipo de autenticación donde se corrobora que una entidad es la fuente original de la creación de un conjunto de datos en un tiempo específico. Por definición, la *autenticación de origen* incluye la integridad de datos, pues cuando se modifican los datos, se tiene una nueva fuente. 1, 38, 63, *véase también* INTEGRIDAD DE DATOS.

3. Autenticación multifactor

(*Multi-factor authentication*) Método de autenticación que requiere al menos dos métodos (independientes entre sí) para identificar al usuario. 1, 54, 61

4. Autenticación mutua

(*Mutual authentication*) Autenticación en la cual cada una de las partes identifica a la otra. 1, 54

5. Biyección

Dicho de las funciones que son inyectivas y suprayectivas al mismo tiempo; en otras palabras, que todos los elementos del conjunto de salida tengan una imagen distinta en el conjunto de llegada y a cada elemento del conjunto de llegada le corresponde un elemento del conjunto de salida. 1, 13, *véase también* INYECTIVA, SUPRAYECTIVA, FUNCIÓN y IMAGEN.

6. Cifrado con prefijo

Técnica de ordenamiento pseudoaleatorio que consite en seguir el orden dado por el texto cifrado de todos los elementos a ordenar. 1, 47

7. Cifrado de caminata cíclica

(*Cycle-walking cipher*) Método diseñado para cifrar mensajes de un espacio M utilizando un algoritmo de cifrado por bloques que actúa en un espacio $M' \supset M$ y obtener textos cifrados que están en M al cifrar iterativamente hasta que el mensaje cifrado se encuentra en el dominio deseado. 1, 103

8. Cifrado iterativo

(*Iterated block cipher*) Cifrado de bloque que involucra la repetición secuencial de una función interna llamada función de ronda. Los parámetros incluyen el número de rondas, el tamaño de bloque y el tamaño de llave. 1, 10, *véase también* RONDA.

9. Circuito booleano

(*Boolean circuit*) Modelo matemático definido en términos de compuertas lógicas digitales (AND, OR, NOT, etc.). 1

10. Codominio

Una función mapea a los elementos de un conjunto A con elementos de un conjunto B; A es el dominio y B es el *codominio*. 1, *véase también* FUNCIÓN y DOMINIO.

11. Computacionalmente indistinguible

(*Computational indistinguishability*) Para un A tomado de algún conjunto de distribución y un circuito booleano C (con las suficientes entradas), p_C^A es la probabilidad de que la salida del circuito booleano C sea 1 para una entrada de A. Se dice que los conjuntos de distribución $\{A_k\}$ y $\{B_k\}$ son *computacionalmente indistinguibles* si para cualquier familia de circuitos de tamaño polinomial $C = \{C_k\}$, la función $e(k) = |p_{A_k}^{C_k} - p_{B_k}^{C_k}|$ es despreciable [41].
Otra forma de expresarlo, en un contexto más criptográfico, es como la incapacidad de un adversario de distingir si la salida de una primitiva criptográfica es una permutación aleatoria o una permutación pseudoaleatoria: una permutación P es segura (en términos de un ataque de texto cifrado conocido) cuando es *computacionalmente indistingible* de una permutación aleatoria. 1, 71, *véase también* FUNCIÓN, FUNCIÓN DESPRECIABLE, CONJUNTO DE DISTRIBUCIÓN y CIRCUITO BOOLEANO.

12. Computacionalmente no factible

(*Computationally infeasible*) Se dice que una tarea es *computacionalmente no factible* si su costo (medido en términos de espacio o de tiempo) es finito pero ridículamente grande. 1, 53

13. Conjunto de distribución

(*Distribution ensemble*) Un *conjunto de distribución* $\{A_k\}$ es una familia de medidas de probabilidad en $\{0,1\}^*$ para la cual hay un polinomio q tal que las únicas cadenas de longitud mayor a $q(k)$ tienen una probabilidad distinta de cero en $\{A_k\}$ [41]. 1

14. Criptografía fuerte

(*Strong cryptography*) De acuerdo al PCI SSC (en [42]), es la criptografía basada en algorítmos probados y aceptados en la industria, junto con longitudes de llaves fuertes y buenas prácticas de administración de llaves. 1, 52, 54, 66

15. Criptología

Estudio de los sistemas, claves y lenguajes secretos u ocultos. 1, 4, 5

16. Distribución de probabilidad

Una distribución de probabilidad P en el conjunto de eventos S es una secuencia de números positivos p_1, p_2, $\ldots$ p_n que sumados dan 1. Donde p_i se interpreta como la probabilidad de que el evento s_i ocurra. 1, 87

17. Distribución uniforme

Distribución de probabilidad en la que todos los elementos tienen la misma probabilidad ocurrencia. 1, 75, 86, *véase también* DISTRIBUCIÓN DE PROBABILIDAD.

18. Dominio

El *dominio* de una función $f(x)$ es el conjunto de valores para los cuales la función está definida. 1, *véase también* FUNCIÓN y CODOMINIO.

19. Entropía

Definida para una función de probabilidad de distribución discreta, mide cuánta información en promedio es requerida para identificar muestras aleatorias de esa distribución. 1, 10, 55, 69, 70, 76–79, 81, 84–86, *véase también* FUNCIÓN y DISTRIBUCIÓN DE PROBABILIDAD.

20. Equiprobable

Se dice que un conjunto de eventos es equiprobable cuando cada uno tiene la misma probabilidad de ocurrencia. 1, 57, 101

21. Estadísticamente independiente

Dicho de la ocurrencia de dos eventos E_1 y E_2: si $P(E_1 \cap E_2) = P(E_1)P(E_2)$ entonces E_1 y E_2 son *estadísticamente independientes* entre sí. Es importante notar que si esto ocurre, entonces $P(E_1|E_2) = P(E_1)$ y $P(E_2|E_1) = P(E_2)$, es decir, la ocurrencia de uno no tiene ninguna influencia en las probabilidades de ocurrencia del otro. 1, 57, 58, *véase también* PROBABILIDAD CONDICIONAL.

22. Fuerza efectiva

Para un espacio de llaves K, su *fuerza efectiva* es $\log_2 |K|$ (el logaritmo base dos de su cardinalidad). 1, 57, 59, 65

23. Función

Regla entre dos conjuntos A y B de manera que a cada elemento del conjunto A le corresponda un único elemento del conjunto B. 1, *véase también* CODOMINIO y DOMINIO.

24. Función booleana

Son las funciones que mapean f a un valor del conjunto booleano $0, 1$ o *verdadero* y *falso*. Formalmente, se define como $f : B^n \to B$, donde $B = 0, 1$ y n un entero no negativo que corresponde al número de argumentos, o variables, que necesita la función. 1, 102, *véase también* FUNCIÓN.

25. Función despreciable

(*Negligible function*) Una función $e : N \to R$ es *despreciable* si, para todos los enteros positivos c, existe un entero N_c tal que para todo $x \geq N_c$, $|e(x)| \leq \frac{1}{x^c}$. Esto significa que $e(x)$ se desvanece más rápido que el inverso de cualquier polinomio [41]. 1, *véase también* FUNCIÓN.

26. Imagen

Suponga que se tiene $x \in X$ y $y \in Y$ tal que $f(x) = y$; se dice entonces que y es la *imagen* de x bajo f, o que x es preimagen de y. 1, *véase también* PREIMAGEN, FUNCIÓN, DOMINIO y CODOMINIO.

27. Integridad de datos

Propiedad en la que los datos no han sido alterados sin autorización desde que fueron creados, transmitidos o almacenados por una fuente autorizada. Operaciones que insertan, eliminan, modifican o reordenan bits invalidan la *integridad de los datos*. La *integridad de los datos* incluye que los datos estén completos y, cuando los datos son divididos en bloques, cada bloque debe cumplir con lo mencionado anteriormente. 1, 38

28. Inyectiva

Una función $f : D_f \to C_f$ es *inyectiva* (o uno a uno) si a diferentes elementos del dominio le corresponden diferentes elementos del codominio; se cumple para dos valores cualesquiera $x_1, x_2 \in D_f$ que $x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$. 1, *véase también* FUNCIÓN, DOMINIO, CODOMINIO, BIYECCIÓN y SUPRAYECTIVA.

29. Libreta de un solo uso

(*One-time pad*) Algorimo de cifrado donde el texto en claro se combina con una llave secreta que es, al menos, de la longitud del mensaje. Si es utillizado e implementado correctamente, este algoritmo es indescifrable. 1, 41

30. Material de llaves

(*keying material*) Conjunto de llaves criptográficas sin un formato específico. 1, 71, 72, 75, 76

31. Modo de operación

Construcción que permite extender la funcionalidad de un cifrado a bloques para operar sobre tamaños de información arbitrarios. 1, 9, 23–28, 45, 59, 128, 130, 131

32. Máquina de Turing

(*Turing machine*) Se considera como una cinta infinita dividida en casillas, cada una de las cuales contiene un símbolo. Sobre dicha cinta actúa un dispositivo que puede adoptar distintos estados y que, en cada instante, lee un símbolo de la casilla sobre la que está situado; dependiendo del símbolo leído y del estado en el que se encuentra, la máquina realiza las siguientes tres acciones: primero, pasa a un nuevo estado; segundo, imprime un símbolo en el lugar del que acaba de leer; y, tercero, se desplaza hacia la derecha, hacia la izquierda o se detiene. 1

33. Nonce

Valor que varía con el tiempo y es improbable que se repita. Por ejemplo, puede ser un valor aleatorio generado para cada uso, una etiqueta de tiempo, un número de secuencia o una combinación de los tres. 1, 76, 78, 79

34. Oráculo

Oracle machine Se refiere a una máquina abstracta utilizada para estudiar problemas de decisión. Puede verse como una máquina de Turing con una caja negra (llamada *oráculo*) que puede resolver ciertos problemas de decisión u obtener el valor de una función en una sola operación. 1, 56, *véase también* MÁQUINA DE TURING.

35. Permutación

Sea S un conjunto finito de elementos. Una *permutación* p en S es una biyección de S a sí misma (i. e. $p : S \to S$). 1, 5, 56, 58, *véase también* BIYECCIÓN.

36. Preimagen

Suponga que se tiene $x \in X$ y $y \in Y$ tal que $f(x) = y$; se dice entonces que x es *preimagen* de y, o que y es la imagen de x bajo f. 1, 34, 35, 82, *véase también* IMAGEN, FUNCIÓN, DOMINIO y CODOMINIO.

37. Primitiva criptográfica

(*Cryptographic primitive*) Algoritmos criptográficos que son usados con frecuencia para la construcción de protocolos de seguridad. En [1] se clasifican en tres categorías principales: de llave simétrica, de llave pública y sin llave. 1, 52, 56, 59, 91, 92, 100, 101, 103

38. Probabilidad condicional

Sean E_1 y E_2 dos eventos, con $P(E_2) \geq 0$. La *probabilidad condicional* se denota por $P(E_1|E_2)$, y es igual a

$$P(E_1|E_2) = \frac{P(E_1 \cap E_2)}{P(E_2)}$$

Esto mide la probabilidad de que ocurra E_1 sabiendo que ya ocurrió E_2 1

39. Registro de desplazamiento

Es un arreglo de *flip-flops* que se encarga de desplazar la información contenida en su estado actual, teniendo una entrada de un bit, para reemplazar el valor del primer *flip-flop* del arreglo. 1

40. Registro de desplazamiento con retroalimentación lineal

(*Linear-feedback shift register (LFSR)*) Es un registro de desplazamiento en el que su bit de entrada es una función lineal de su estado anterior. 1, 89, *véase también* REGISTRO DE DESPLAZAMIENTO.

41. Ronda

(*Round*) Bloque compuesto por un conjunto de operaciones que es ejecutado múltiples veces. Las *rondas* son definidas por el algoritmo de cifrado. 1, 10, 12, 13, 15, 18, 19, 21, 22

42. Semilla

(*Seed*) Cadena de bits que es utilizada como entrada para los los mecanismos DRBG. Determina una parte del estado interno del DRBG. 1, 71, 76–79, 81, 82, 84, 86

43. Suprayectiva

Una función $f : D_f \to C_f$ es *suprayectiva* si todo elemento de su codominio C_f es imagen de por lo menos un elemento de su dominio D_f: $\forall b \in C_f \ \exists a \in D_f$ tal que $f(a) = b$. 1, *véase también* FUNCIÓN, DOMINIO, CODOMINIO, IMAGEN, BIYECCIÓN y INYECTIVA.

44. Tiempo polinomial

Se dice que una función computable o algoritmo es de *tiempo polinomial* cuando su complejidad está, en el peor de los casos, acotada por arriba por un polinomio sobre el tamaño de sus variables de entrada. Si se toma a f como una función computable, esta es de *tiempo polinomial* si $f \in O(n^k)$ donde $k \geq 1$. 1, 71

45. Token

Valor representativo que se usa en lugar de información valiosa. 1, 52–58, 60, 61, 90–92, 100–102, 105, 106, 110, 129, 132

46. Vector de inicialización

Cadena de bits de tamaño fijo que sirve como entrada a muchas primitivas criptográficas (p. ej. algunos modos de operación). Generalmente se requiere que sea generado de forma aleatoria. 1, 24, 26, 30, 44, 45, 64, 72, 97, *véase también* MODO DE OPERACIÓN y PRIMITIVA CRIPTOGRÁFICA.

Siglas y acrónimos

Criptográficos

ABL Arbitrary Block Length Mode. 44

AES Advanced Encryption Standard. 7, 15, 33, 36, 46, 55, 59, 93, 94, 110

CAVP Cryptographic Algorithm Validation Program. 59

CBC Cipher-block Chaining. 23–25, 28, 38, 44, 59, 94, 124, 128, 131

CFB Cipher Feedback. 24, 26, 28, 59, 128, 131

CMAC Cipher-based MAC. V, 41, 71, 75, 128

CMC CBC-Mask-CBC. 44

CRHF Collision-Resistant Hash Function. 35

CSP Critical Security Parameter. 78, 79

CTR Counter Mode. 28, 45, 59, 125, 131

DES Data Encryption Standard. 6, 13, 15, 18, 38, 59, 126

DH Diffie-Hellman. 59, 124

DHE DH Ephemeral. 59

DRBG Deterministic Random Bit Generator. 76–79, 82, 84, 85, 91, 92, 104, 106, 110, 123, 132

DSA Digital Signature Algorithm. 59, 124

ECB Electronic Codebook. 23, 24, 44, 124, 128, 131

ECC Elliptic Curve Cryptosystem. 59

ECDH Elliptic Curve DH. 59

ECDSA Elliptic Curve DSA. 59

ECIES Elliptic Curve Integrated Encryption Scheme. 59

ECMQV Elliptic Curve Menezes-Qu-Vanstone. 59

EME ECB-Mask-ECB. 44

FEAL Fast Data Encipherment Algorithm. 18

FFX Format-preserving Feistel-based Encryption. 92, 93, 130

FPE Format Preserving Encryption. 46, 100

HCTR Hash CTR. 45

HMAC Keyed-Hashed Message Authentication Code. V, 41, 43, 71, 75, 128

IDEA International Data Encryption Algorithm. 19

KDF Key Derivation Function. 71, 75, 76

MAA Message Authenticator Algorithm. 38

MAC Message Authentication Code. V, 35, 38, 39, 41, 94, 124, 125, 131

MD4 Message Digest-4. 36, 38, 59

MD5 Message Digest-5. 35, 36, 59

MDC Message Digest Cipher. 38

MDC-2 Modification Detection Code-2. 35

NMAC Nested MAC. V, 39–41, 128

NRBG Non-deterministic Random Bit Generator. 76, 92

OAEP Optimal Asymmetric Encryption Padding. 59

OCB Offset Codebook. 59

OFB Output Feedback. 27, 28, 30, 59, 128, 131

OMAC One-key MAC. V, 39, 41

OWHF One-Way Hash Function. 34, 35, 41

PMAC Parallel MAC. V, 41, 42, 128

PRF Pseudorandom Function. V, 38, 39, 41, 71, 72, 75, 131

PRNG Pseudorandom Number Generator. 58, 85, 86, 88

RBG Random Bit Generator. VI, 69–71, 76

RNG Random Number Generator. 85–88

RSA Ron Rivest, Adi Shamir, Leonard Adleman. 7, 8, 31, 33, 35, 59, 126, 131

RSAES RSA Encryption Scheme. 59

SAFER Secure And Fast Encryption Routine. 21

SHA Secure Hash Algorithm. 35–37, 60, 94

TBC Tweakable Block Chaining. 44, 45, 128

TBC Tweakable Block Cipher. 44, 45

TDES Triple DES. 59, 94

TES Tweakable Encyphering Scheme. 44, 45

TRNG True Random Number Generator. 101

XCB Extended Codebook. 45

Computacionales

API Application Program Interface. 54

ASCII American Standard Code for Information Interchange. 35, 46

DAO Data Access Object. 110

LAN Local Area Network. 31

SSH Secure Shell. 43

SSL Secure Sockets Layer. 31, 35, 43

TLS Transport Layer Security. 31

UML Unified Modeling Language. 105

WEP Wired Equivalent Privacy. 31, 32

WPA WiFi Protected Access. 31

Bancarios

BIN Bank Identifier Number. 102

CDV Card Data Vault. 53, 57–59, 100, 108

DSS Data Security Standard. 52, 54

INN Issuer Identification Number. 48

MII Major Industry Identifier. 48, 49, 130

PA Payment Application. 54

PAN Personal Account Number. 46, 48, 52–54, 56–61, 91, 92, 100–102, 106, 110

PCI Payment Card Industry. VIII, 52, 54, 57, 59–62, 90–92, 100, 119, 130

De instituciones y asociaciones

ECRYPT European Network of Excellence in Cryptology. 32

FIPS Federal Information Processing Standard. 13, 53, 57, 59, 69

IEC International Electrotechnical Commission. 55

IEEE Institute of Electrical and Electronic Engineers. 31

ISO International Organization for Standaridization. 55, 57, 59

NESSIE New European Schemes for Signatures, Integrity and Encryption. 33

NIST National Institute of Standards and Technology. 15, 36, 37, 41, 55, 57–60, 63, 66, 88, 92, 103, 113

NSA National Security Agency. 36

NVLAP National Voluntary Laboratory Accreditation Program. 84

RAE Real Academia Española. 4

SSC Security Standard Council. VIII, 52, 57, 59, 60, 62, 90–92, 119, 130

Lista de figuras

Lista de tablas

Lista de pseudocódigos