Desarrollo de una aplicación MATLAB para generación de vectores test y chequeo de funcionalidad en algoritmos criptográficos de clave simétrica

Autor: Rodrigo Rico Gómez
Tutores: Antonio J. Acosta Jiménez y Erica Tena Sánchez
Departamento de Electrónica y Electromagnetismo
Facultad de Física
Universidad de Sevilla

26 de noviembre de 2019

Índice

1. Introducción

1.1. Objetivo del trabajo

Este trabajo tiene como objetivo el diseño en MATLAB de una aplicación que permita testear algoritmos criptográficos para su implementación física.
Para ello, se ha implementado en MATLAB un cifrador ligero como es *Piccolo*, del cual no existía versión previa en MATLAB, con el objetivo de probar el funcionamiento de la aplicación sobre él.
Se pretende, que la aplicación desarrollada sirva para testear cualquier cifrador de este tipo, siempre que haya sido implementado previamente en MATLAB.
Podemos, por tanto, remarcar dos objetivos principales que tendría este trabajo:

- La implementación en MATLAB del cifrador.
- El desarrollo de una aplicación capaz de testear un cifrador de este tipo y guardar los datos de entrada y salida para su posterior uso.

1.2. Motivación

La implementación física de cualquier algoritmo es un proceso que requiere siempre del posterior testeo del *hardware* que se ha diseñado en el laboratorio. Es importante comprobar que dicho *hardware* responde perfectamente a las especificaciones.
En el caso particular de los algoritmos criptográficos de clave simétrica, cuya función consiste en cifrar un texto utilizando una clave conocida, el testeo consiste en verificar que, para un mismo texto de entrada y clave, se genera el mismo texto cifrado utilizando el *software* y el *hardware* del algoritmo.
La función de la aplicación que se pretende diseñar es la de generar una tabla con distintos valores de entrada y salida del algoritmo implementado en software, en nuestro caso MATLAB. Dicha tabla se utilizará para comprobar que la implementación en hardware genera los mismos datos de entrada y salida.
Uno de los posibles ámbitos de aplicación de esta herramienta es el entorno de ataques con el que se trabaja en el IMSE (Instituto de Microelectrónica de Sevilla), donde existe una línea de investigación dedicada al desarrollo de *hardware* para criptografía.
En este contexto, una herramienta que, a partir de un algoritmo implementado en MATLAB (*software*), sea capaz de generar sucesivos vectores test y, además, guardar el resultado de cada test en un archivo, servirá para comprobar el correcto funcionamiento de los dispositivos microelectrónicos de seguridad (*hardware*), por comparación de los datos de entrada y salida entre ambos.

1.3. Metodología

El procedimiento seguido para la realización del trabajo se ha basado fundamentalmente en la escritura de las distintas partes que componen tanto el cifrador como la aplicación de testeo. La única herramienta utilizada durante el

proceso ha sido, por tanto, el programa MATLAB (en su versión R2013a).
En lugar de aglutinar toda la escritura del algoritmo *Piccolo* en un mismo *script*, se ha decidido, por cuestiones de organización del código, dividirlo en partes con su *script* independiente. Siendo el algoritmo final el resultado de ensamblar estas partes de forma correcta.
A su vez, en ocasiones se ha necesitado crear manualmente subfunciones utilizadas para implementar alguna orden del algoritmo que requería de mayor elaboración. Se ha realizado de esta forma para no ensuciar excesivamente el código y así poder mantener la estructura del algoritmo en la escritura.
La cronología ideal de trabajo hubiera sido implementar completamente el cifrador en MATLAB, comprobar su correcto funcionamiento y posteriormente desarrollar la aplicación. Sin embargo, dada la complejidad de alguna de las partes de *Piccolo*[1], su implementación se ha demorado considerablemente respecto a lo esperado a priori. Por consiguiente, se ha decidido diseñar la aplicación de forma paralela a la escritura del cifrador. Lo cual no ha supuesto problema alguno ya que el funcionamiento de la aplicación es independiente del cifrador.
Una vez se tienen *Piccolo* al completo y la aplicación lista en el entorno de MATLAB, se procede a ejecutar la aplicación sobre el cifrador y se comprueba que se recogen los datos de entrada y salida correctamente.

1.4. Estructura de la memoria

Se puede dividir esta memoria en cuatro partes fundamentales:

- Introducción a la criptografía en clave simétrica: contiene el contexto científico donde se enmarca el trabajo: la criptografía. El objetivo de este apartado no es el de convertir al lector en un experto en criptografía, si no el de presentar una serie de nociones básicas que faciliten el entendimiento del resto de la memoria. En consecuencia, el carácter de esta introducción es eminentemente superficial, explicando los principales conceptos de la criptografía en clave simétrica, sin atender a aspectos técnicos.

- Especificaciones de *Piccolo*: contiene la estructura matemática del cifrador [2], que supone el conocimiento de partida para trasladar a MATLAB una versión funcional del mismo. Adjunto a las especificaciones matemáticas, se exponen brevemente las prestaciones de *Piccolo* como cifrador por bloques ultraligero.

- La implementación en MATLAB de *Piccolo*: contiene explicaciones a cerca de la escritura y la función de cada uno de los *scripts* que componen el cifrador. Están acompañadas de líneas de pseudocódigo[3], que facilitan al lector el entendimiento de la versión del algoritmo en MATLAB.

[1] Las partes de Piccolo que han requerido mayor esfuerzo han sido la expansión de la clave y, sobre todo, la matriz de difusión. Ambas se expondrán detalladamente más adelante.
[2] La estructura matemática ha sido obtenida al completo de la referencia [2].
[3] No se expone el código original dentro de la memoria para evitar ensuciar la escritura de la misma. El código se encuentra en el anexo al final de la memoria.

- El desarrollo de la aplicación de testeo: donde se exponen las prestaciones de esta aplicación y se adjuntan, al igual que en el apartado anterior, líneas de pseudocódigo.

2. Introducción a la criptografía en clave simétrica

Llamamos criptografía a la disciplina encargada de elaborar sistemas para la transmisión segura de información sensible frente a adversarios que buscan acceder a su contenido sin autorización.
Una de las opciones para realizar esta transmisión de forma segura es utilizando un sistema de cifrado. En un sistema de cifrado, el emisor encripta o cifra el mensaje en base a una clave, el mensaje encriptado se envía al receptor, el cual a través de otra clave puede desencriptar o descifrar el mensaje devolviéndolo a su forma original. Si para cifrar y descifrar se usa la misma clave, se dice que el sistema es de clave simétrica.
Los componentes fundamentales de un sistema de cifrado de clave simétrica son los siguientes [1]:

- **Texto plano** (*plaintext*): mensaje en su versión original inteligible que se desea enviar al receptor o receptores de forma segura. Esta secuencia de datos es una de las dos entradas que alimenta el algoritmo de encriptado.
- **Clave secreta** (*key*): es una secuencia de datos independiente que sirve para encriptar el texto plano. Este dato debe permanecer secreto entre los destinatarios del mensaje.
- **Algoritmo de encriptado**: estructura matemática que aplica sucesivas sustituciones y permutaciones sobre los integrantes del texto plano utilizando la clave secreta para generar una versión ininteligible del mensaje. Las entradas del algoritmo serían, por tanto, el texto plano y la clave secreta, y la salida el texto cifrado.
- **Texto cifrado** (*ciphertext*): texto ilegible salida del algoritmo de encriptado cuya forma depende tanto del *plaintext* como de la clave. Aparentemente es una secuencia aleatoria de datos.
- **Algoritmo de desencriptado**: función inversa al algoritmo de encriptado. Toma la clave y el *ciphertext* y recompone el *plaintext*.

El algoritmo implementado en este trabajo es un algoritmo de encriptado, lo cual significa que su función es cifrar información, es decir, convertir el *plaintext* o mensaje legible en el *ciphertext* o mensaje ilegible utilizando para ello la clave.

Se pueden observar en la Figura 1[4] los principales componentes del sistema

[4]Este diagrama ha sido realizado utilizando la herramienta Lucidchart.

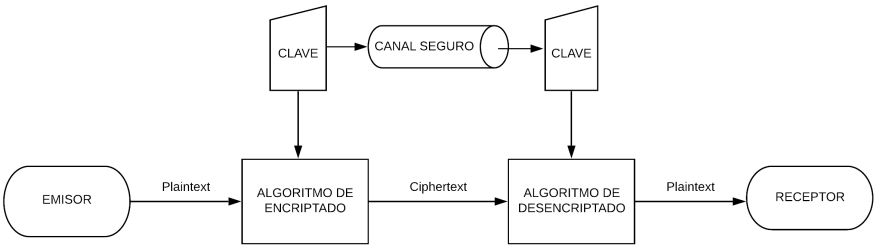

Figura 1: Esquema básico de un sistema de cifrado simétrico convencional.

de cifrado simétrico. El sistema que se va a implementar en MATLAB en este trabajo es el cifrador o algoritmo de encriptado únicamente, cuyas entradas serían el *plaintext* y la clave y obtendría como salida el *ciphertext*. Esto empieza a dislumbrar una idea de la forma que tendrá el *script* de *Piccolo* en MATLAB: debe ser en última instancia una función de dos entradas y una salida, lo cual no significa que no esté compuesta por varias subfunciones ensambladas.

2.1. La seguridad de un sistema de cifrado simétrico

Cualquier sistema de cifrado simétrico necesita cumplir dos requerimientos importantes para ser considerado seguro:

1.Necesitamos un sistema de encriptado fuerte. Un adversario que conozca completamente el algoritmo y que además tenga acceso a varias parejas de datos *plaintext-ciphertext* debe ser incapaz de averiguar la clave secreta [1].

2.Como consecuencia del punto anterior, hemos confiado la seguridad del sistema en la confidencialidad de la clave, ya que estamos suponiendo que el adversario conoce el algoritmo. Un buen sistema de cifrado simétrico debe, por consiguiente, ir siempre acompañado de un canal seguro [1] a través del cual transmitir la clave a los receptores.

En la mayoría de los casos, por tanto, el atacante está interesado en averiguar la clave más que e descifrar el mensaje [1], ya que en la clave está la llave para acceder a todos los mensajes enviados. Un sistema de cifrado simétrico del cual se conozca el algoritmo de encriptado puede ser más o menos vulnerable, si además se conocen parejas de datos *plaintext-ciphertext*, estaría más expuesto aún a ataques. Sin embargo, con una clave lo suficientemente fuerte se puede lograr una seguridad aceptable. En otras palabras, si cae la clave cae el sistema. La clave es, en consecuencia, el arma más poderosa que tiene el usuario y el bien más codiciado por el atacante.

Un adversario tiene dos opciones para averiguar la clave: realizar un criptoanálisis o un ataque por fuerza bruta [1].

- **Criptoanálisis**: se basan en analizar la estructura del algoritmo más, quizás, el conocimiento de parejas de datos entrada-salida [1]. Este tipo de ataque explota las características del algoritmo para deducir la clave.

- **Ataque por fuerza bruta**: el atacante intenta desencriptar el *ciphertext* utilizando todas las posibles claves hasta que una de ellas devuelve un *plaintext* legible.

Un sistema de cifrado es incondicionalmente seguro si: es seguro contra un adversario que dispone de infinito tiempo e infinito poder computacional para deducir la clave, lo cual es imposible en el mundo real [1]. De manera que, ante la imposibilidad de crear un sistema de estas características, la seguridad de un sistema de cifrado simétrico siempre está sujeta al cumplimiento de uno de los siguientes dos factores:

- El coste que supone acceder a la información cifrada es superior al valor de la información en sí [1]. Ésta es una cuestión de rentabilidad, el atacante debe evaluar si le merece la pena el esfuerzo de atacar el sistema. Es lógico, por tanto, suponer, que cuanto más valiosa sea la información que se intenta transmitir, más fuerte debe ser el sistema de cifrado.

- El tiempo requerido para averiguar la clave es superior al tiempo de vida útil de la información a descifrar [1] o de la misma clave. Debido a esto, para ganar en seguridad, siempre es aconsejable cambiar de clave cada cierto tiempo.

El algoritmo *Piccolo* soporta, en la versión que se va a implementar en este trabajo, una clave de 80 bits [5], lo cual significa que existen 2^{80} posibles claves. Un conjunto de varias máquinas cooperando tardaría suficiente tiempo en encontrar la clave como para considerar el sistema de cifrado seguro. Si, además, la clave se cambia cada cierto tiempo, sería aún más seguro.
El caso particular del algoritmo que se va a implementar, *Piccolo-80*, ha sido evaluado frente a varios métodos de criptoanálisis [5], entre los que destacan el *Meet-In-The middle attack* (MITM) en [8], el ataque *biclique* en [7], o el diferencial imposible (*impossible differential*) [6].
También ha sido enfrentado a uno de los ataques más poderosos, de criptoanálisis lineal [5], introducido por Matsui en [9], lo cual da cuenta del nivel de seguridad de *Piccolo*. Como dato comparativo, cabe reseñar que una versión mejorada de este tipo de ataques trató de averiguar la clave del DES (*Data Encryption Standard*) de 16 rondas [10] con éxito.

2.2. Tipos y características de los sistemas de cifrado

Podemos caracterizar sistemas de cifrado en base a una serie de aspectos fundamentales:

- El **tipo de operaciones** utilizadas por el algoritmo de encriptado para transformar el *plaintext* en *ciphertext*. Todos los algoritmos suelen estar basados es dos operaciones fundamentales [1]

[5] Existe una versión de Piccolo que trabaja con una clave de 128 bits, en este trabajo se va a implementar sin embargo su versión de 80 bits.

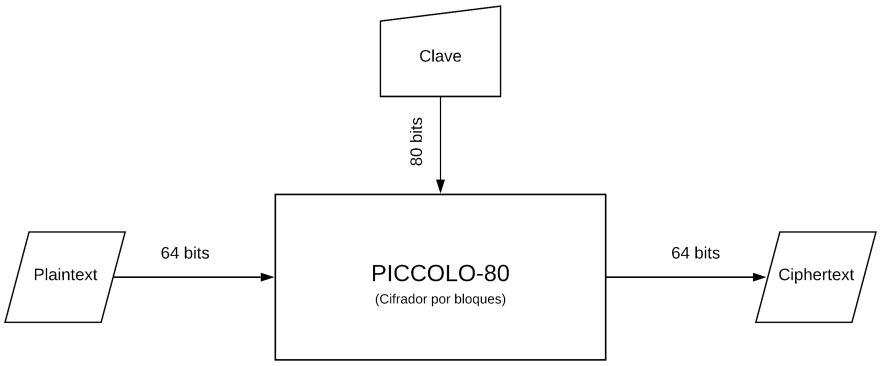

Figura 2: Entradas y salida del algoritmo Piccolo-80.

- Sustitución: cada elemento del *plaintext* (bit, letra, bloque...) es reemplazado por otro elemento distinto.
- Permutación: los elementos del *plaintext* son reordenados.

- El **número de claves** utilizado: si el emisor y el receptor emplean la misma clave para cifrar y descifrar, se dice que el sistema es, como se ha dicho antes, de clave simétrica o clave secreta. Si utilizan distinta clave, se denomina de clave asimétrica o clave pública.
- El modo en el que se **procesan los datos**: un cifrador por boques (*block cipher*) procesa la entrada dividida en bloques de un número determinado de elementos. Toma un bloque, lo cifra, y luego toma el siguiente bloque. Así hasta cifrar la totalidad del *plaintext*. Por otro lado están los *stream ciphers* que procesan el texto de forma continua.

El algoritmo de encriptado implementado en este trabajo, *Piccolo*, cifra el texto por bloques de 64 bits y su procesamiento de datos tiene estructuras tanto de sustitución como de permutación. Además, como se ha comentado anteriormente, es de clave simétrica, emplea la misma clave de 80 bits para encriptar y desencriptar el mensaje. Con estos datos, se puede deducir que nuestro *script* de MATLAB tendrá una entrada de 64 bits, otra entrada de 80 bits y una salida de 64 bits. Se puede visualizar mejor en el esquema de la Figura 2[6].

2.3. Cifradores por bloques (*block ciphers*)

Un cifrador por bloques es un sistema de cifrado/descifrado que toma un bloque del *plaintext* como un todo, opera sobre él, y genera un bloque de *ciphertext* de la misma longitud [1]. De esta forma, dividiendo la tarea en bloques, el algoritmo encripta la totalidad del mensaje.

[6]Este diagrama ha sido realizado utilizando la herramienta Lucidchart.

Muchos cifradores por bloques usan **estructuras de Feistel**. Estas estructuras consisten en la ejecución de un número determinado de rondas idénticas de procesamiento. Es decir, aplicar la misma operación de manera sucesiva un determinado número de veces [1]. A cada una de ellas se le llama ronda, y éstas están formadas por una operación de sustitución y otra de permutación. Al final, la estructura de Feistel es la forma que tienen la mayoría de los cifradores por bloques de aplicar estas dos operaciones criptográficas fundamentales (sustitución y permutación).
Una condición importante que deben cumplir los cifradores por bloques simétricos, como es el caso del algoritmo *Piccolo*, es la generación unívoca de un bloque de *ciphertext* a partir de un bloque de *plaintext* [1]. Esto quiere decir, que la transformación debe ser reversible para que, a desencriptar el mensaje cifrado, se obtenga siempre el mensaje original.

3. *Piccolo*: características y especificaciones

Piccolo pertenece a una rama de la critografía que se denomina criptografía ultraligera (*Lightweight Cryptography*) o *LWC* [11]. La criptografía ultraligera investiga la integración de estructuras criptográficas primitivas y algoritmos para su implementación en dispositivos micrométricos [11].
Como se ha mencionado anteriormente, *Piccolo* es un cifrador por bloques de implementación física ligera, que soporta claves de 80 bits, aunque modificando ligeramente el algoritmo se puede conseguir que acepte claves de 128 bits. Los bloques de trabajo son de 64 bits. Esta estructura de entrada y salida de información se aprecia perfectamente en la Figura 2.
Con *Piccolo* se propone un sistema de cifrado por bloques que alcanza grandes prestaciones en cuanto a su ligereza a la hora de ser implementado en *hardware*, sin comprometer su seguridad, que también es remarcable.
La versión convencional (claves de 80 bits) gana con respecto a la versión de 128 bits en cuanto a ligereza. La primera es implementable con 683 puertas, mientras que la segunda supone 758 [2]. Por otra parte, en el campo de la seguridad, la clave de 128 bits se hace mucho menos vulnerable a ataques.
Piccolo también alcanza muy buenas prestaciones de eficiencia energética. Reduciendo considerablemente lo que se conoce como el consumo de energía por bit [2] con respecto a otros cifradores por bloques.
Todas estas características lo hacen un excelente candidato para aplicaciones de bajo consumo energético y sistemas micrométricos, como etiquetas RFID o sensores [2].

3.1. Estructura general del algoritmo

Al igual que la mayoría de los cifradores por bloques, la estructura del algoritmo se puede dividir en dos partes fundamentales. Cada una de las cuales cumple una función totalmente distinta:

- **Expansión de la clave**: su función consiste en recibir como entrada la

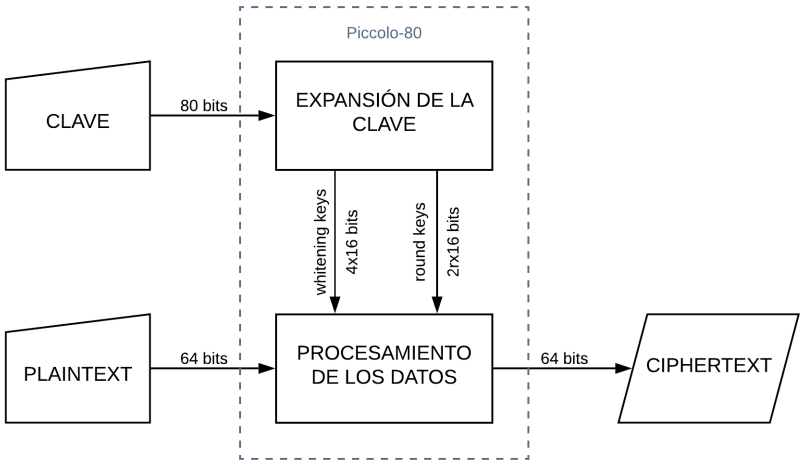

Figura 3: Diagrama de la estructura básica de *Piccolo-80*.

clave de 80 bits y elaborar dos salidas: las *whitening keys* $wk_{i(16)}$ que son 4 subclaves ($0 \leq i < 4$) de 16 bits cada una, y por otro lado las *round keys* $rk_{j(16)}$ que son $2r$ subclaves ($0 \leq j < 2r$) también de 16 bits cada una, siendo r un parámetro de entrada del algoritmo, que indica el número de rondas que trabaja la estructura de procesamiento de datos. Para la versión *Piccolo-80* (clave de 80 bits), el valor de r debe fijarse en 25 [2].

- **Procesamiento de los datos**: su función es tomar como entrada los datos del *plaintext*, que será un bloque de 64 bits $X_{(64)}$, las 4 subclaves $wk_{i(16)}$ y las $2r$ subclaves de ronda $rk_{j(16)}$, procesar esa información y, en base a ella, construir el bloque de 64 bits de *ciphertext* $Y_{(64)}$.

El ensamblaje de estas dos partes para formar el cifrador se puede consultar de manera visual en el diagrama de la Figura 3. A su vez, el procesamiento de los datos se divide en partes más elementales, que se expondrán matemáticamente y se acompañarán con diagramas en sucesivos apartados de esta sección.

3.2. Expansión de la clave

La expansión de la clave de Piccolo, como se ha explicado antes, tiene la función de preparar las subclaves necesarias para procesar los datos a partir de la clave de entrada de 80 bits. Estas subclaves se dividen en dos tipos: las *whitening keys* que son 4 subclaves de 16 bits, y las *round keys* que son $2r$ subclaves también de 16 bits.
Lo primero que realiza esta parte es dividir la clave de entrada de 80 bits en 5 paquetes de 16 bits: $k_{i(16)} (0 \leq i < 5)$. La generación de las mismas se realiza mediante un algoritmo, el algoritmo de expansión de la clave cuyo símbolo

matemático será a partir de ahora: KS_r, siendo r el número de rondas (25). Su estructura matemática se expone a continuación:

Algortimo $KS_r(K_{(80)})$:
$wk_0 \leftarrow k_0^L \mid k_1^R, wk_1 \leftarrow k_1^L \mid k_0^R, wk_2 \leftarrow k_4^L \mid k_3^R, wk_3 \leftarrow k_3^L \mid k_4^R$
for $i \leftarrow 0{:}(r-1)$

$$(rk_{2i}, rk_{2i+1}) \leftarrow (con_{2i}, con_{2i+1}) \oplus \begin{cases} (k_2 \mid k_3) & \text{si } i \bmod 5 = 0 \text{ ó } 2 \\ (k_0 \mid k_1) & \text{si } i \bmod 5 = 1 \text{ ó } 4 \\ (k_4 \mid k_4) & \text{si } i \bmod 5 = 3 \end{cases}$$

Aparecen en el algoritmo subpaquetes de la clave k_i^L y k_i^R, que no son más que secciones de 8 bits a partir de las cuales se construyen las *whitening keys*. Si un k_i tiene 16 bits de longitud, k_i^L y k_i^R serán los 8 bits de la izquierda y los 8 bits de la derecha respectivamente. Se puede consultar el diagrama de generación de las *whitening keys* en la Figura 4 si se desea una visión más ilustrativa del proceso.
Las constantes (con_{2i}, con_{2i+1}) que aparecen como primer argumento del XOR del bucle, son constantes que dependen del índice i y que son generadas de la siguiente forma:

$$(con_{2i}, con_{2i+1}) \leftarrow (c_{i+1} \mid c_0 \mid c_{i+1} \mid \{00\}_2 \mid c_{i+1} \mid c_0 \mid c_{i+1}) \oplus \{\text{0f1e2d3c}\}_{16}$$

Siendo c_i la representación binaria del índice i en una longitud fija de 5 bits, de manera que, por ejemplo:

$$c_{11} = \{01011\}$$

Como observamos, si el número i no necesita 5 bits en su representación, se rellena con ceros hasta que alcance una longitud de 5 bits, ya que la longitud de los paquetes de bits no puede depender de ninguna variable. Este hecho es aplicable a todas las partes del algoritmo.
En la figura 5 se ilustra el bucle que sigue el algoritmo de expansión de la clave para generar las *round keys*, se recuerda que según las especificaciones r debe ser igual a 25.

3.3. Procesamiento de los datos

Esta parte se encarga básicamente de tomar el *plaintext*, procesarlo y elaborar el *ciphertext* utilizando las subclaves generadas en la expansión de la clave. Al tratarse *Piccolo* de un cifrador por bloques, este proceso se ejecuta sobre un bloque de 64 bits como un todo. La estructura matemática del procesamiento de los datos no es más que la combinación de una serie de operaciones de sustitución y permutación sobre una cadena de 64 bits que dan lugar a otra cadena de 64 bits. Dicha transformación, como se ha señalado anteriormente, debe respetar la reversibilidad del proceso, de manera que se garantice que si aplicamos la función inversa, el descifrado, obtendremos de nuevo el mismo bloque de 64 bits de *plaintext*. Se denota al algoritmo que procesa los datos como G_r:

$$G_r : (X_{(64)}, wk_{i(16)}(0 \leq i < 4), rk_{j(16)}(0 \leq j < 2r)) \longrightarrow Y_{(64)}$$

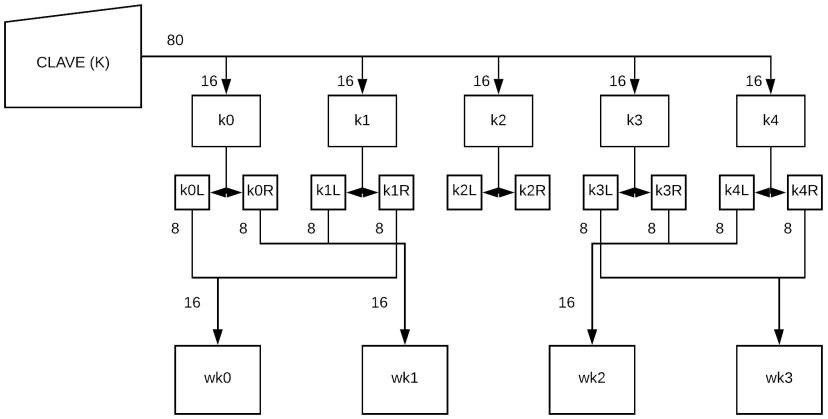

Figura 4: Generación de las *whitening keys* a partir de la clave.

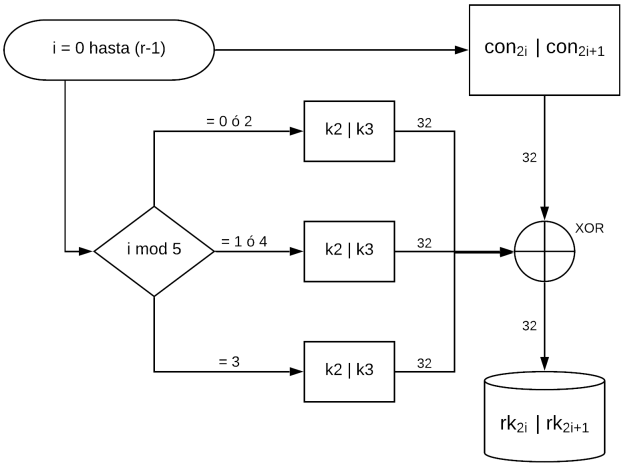

Figura 5: Generación de las *round keys* a partir de la clave y la constante de ronda.

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
S(x)	e	4	b	2	3	8	0	9	1	a	7	f	6	c	5	d

Cuadro 1: Tabla que define la sustitución *S-box* en representación hexadecimal.

Siendo $X_{(64)}$ el bloque de *plaintext* y $Y_{(64)}$ el bloque de *ciphertext*. Las subclaves $wk_{i(16)}$ y $rk_{j(16)}$ ya se definieron en el apartado anterior.
El algoritmo que describe la función G_r es el siguiente:

Algortimo $G_r(X_{(64)}, wk_{i(16)}, rk_{j(16)})$:
$X_{0(16)} \mid X_{1(16)} \mid X_{2(16)} \mid X_{3(16)} \leftarrow X_{(64)}$
$X_0 \leftarrow X_0 \oplus wk_0, X_2 \leftarrow X_2 \oplus wk_1$
for $i \leftarrow 0 : (r-2)$
    $X_1 \leftarrow X_1 \oplus F(X_0) \oplus rk_{2i}$
    $X_3 \leftarrow X_3 \oplus F(X_2) \oplus rk_{2i+1}$
    $X_0 \mid X_1 \mid X_2 \mid X_3 \leftarrow RP(X_0 \mid X_1 \mid X_2 \mid X_3)$
$X_1 \leftarrow X_1 \oplus F(X_0) \oplus rk_{2r-2}$
$X_3 \leftarrow X_3 \oplus F(X_2) \oplus rk_{2r-1}$
$X_0 \leftarrow X_0 \oplus wk_2, X_2 \leftarrow X_2 \oplus wk_3$
$Y_{64} \leftarrow X_0 \mid X_1 \mid X_2 \mid X_3$

Este algoritmo evidentemente por sí solo, sin las funciones que contiene, no sirve aún para cifrar, simplemente establece el esqueleto o estructura sobre el que se cimienta el procesamiento de los datos. Se pude consultar dicha estructura de forma más visual en la Figura 6 [2]. Las dos funciones que aparecen dentro del algoritmo son F y RP.

Función F
Esta función trabaja con argumentos de entrada y salida en representación binaria, ambas de 16 bits de longitud:

$$F : \{0,1\}^{16} \longrightarrow \{0,1\}^{16}$$

Está formada por dos operaciones básicas: una de sustitución (*S-box*) y otra de difusión en forma de matriz (M). La función F es, por tanto, una combinación de estas dos últimas, descrita de forma gráfica en la Figura 7. F es, en definitiva, una matriz de difusión separada por dos barreras de sustitución *S-box*.
Como se puede observar, *S-box* opera sobre paquetes de 4 bits y devuelve como salida otro de la misma longitud. Para actuar sobre la entrada de 16 bits, ésta debe dividirse primero en 4 partes de 4 bits y aplicar a cada uno una operación *S-box*, la cual es simplemente la sustitución de un paquete de 4 bits por otro según el Cuadro 1. La función se define en forma hexadecimal, ya que cada uno de los números o letras que aparecen como argumentos son representables en 4 bits en el sistema binario.

$$(x_{0(4)}, x_{1(4)}, x_{2(4)}, x_{3(4)}) \longleftarrow (S(x_{0(4)}), S(x_{1(4)}), S(x_{2(4)}), S(x_{3(4)}))$$

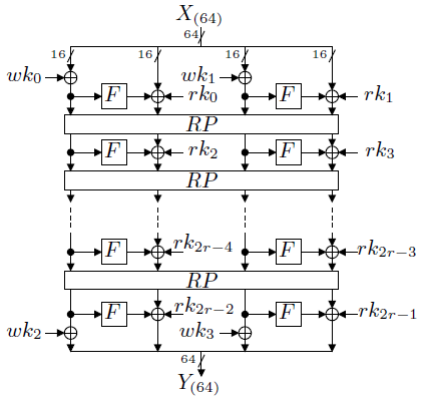

Figura 6: Procesamiento de los datos (encriptado). Obtenida de [2].

Donde $X_{(16)} = x_{0(4)} \mid x_{1(4)} \mid x_{2(4)} \mid x_{3(4)}$. La matriz de difusión opera sobre cuatro paquetes de 4 bits, cada uno representable en forma decimal como un número comprendido entre el 0 y el 15. De manera que opera sobre un vector de 4 componentes y devuelve otro. Es evidente que dicha matriz debe ser cuadrada y de dimensión 4:

$$M = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

También es importante que los números que forman el vector de salida, sean representables en 4 bits, para ser coherentes con el diagrama la Figura 7. La operación de difusión debe, por tanto, conservar el formato de cuatro paquetes de 4 bits.

$$(x_{0(4)}, x_{1(4)}, x_{2(4)}, x_{3(4)})^t \longleftarrow M(x_{0(4)}, x_{1(4)}, x_{2(4)}, x_{3(4)})^t$$

Donde t indica la transposición del vector. Para lograr que los datos de salida $x_{i(4)}$ contengan todos 4 bits se realiza la multiplicación entre el vector y la matriz sobre el campo $GF(2^4)$ utilizando como polinomio irreducible $x^4 + x + 1$. De esta forma se garantiza que al realizar la multiplicación, los datos de salida no excedan el valor de 15, último número decimal representable mediante una secuencia de 4 bits.
Una vez definidas la sustitución y la difusión, solo queda ensamblar ambas operaciones según la Figura 7.

Función RP (*Round Permutation*)
Esta función trabaja sobre una entrada de 64 bits de longitud y devuelve otra

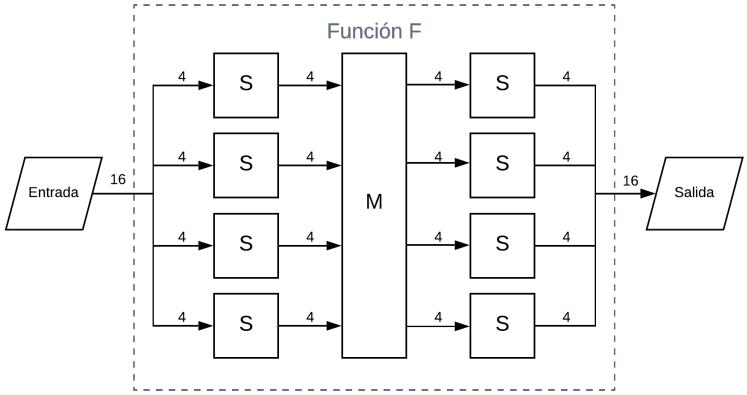

Figura 7: Esquema de la función F. S: sustitución, M: difusión

secuencia de 64 bits. Su función es simplemente dividir los 64 bits en 8 paquetes de 8 bits (1 byte) y permutarlos de la siguiente forma:

$$RP : (x_{0(8)}, x_{1(8)}, ..., x_{7(8)}) \longleftarrow (x_{2(8)}, x_{7(8)}, x_{4(8)}, x_{1(8)}, x_{6(8)}, x_{3(8)}, x_{0(8)}, x_{5(8)})$$

La salida sería la concatenación de $(x_{0(8)}, x_{1(8)}, ..., x_{7(8)})$ en un total de 64 bits.

4. Implementación de *Piccolo* en MATLAB

En esta sección se detallan los procedimientos seguidos para trasladar el cifrador *Piccolo-80* al entorno de MATLAB. El objetivo, como se ha comentado anteriormente, consiste en elaborar un *script* de tipo *function* con dos entradas y una salida. Las entradas serán el bloque de *plaintext* (64 bits) y la clave (80 bits), y la salida el bloque de *ciphertext* (64 bits).
Con el objetivo de facilitar la escritura, el cifrador es un conjunto de *scripts*, cada uno de los cuales desempeña una tarea dentro del algoritmo. Las dos estructuras principales que componen el código son el procesamiento de los datos y la expansión de la clave. A su vez, el procesamiento de los datos hace uso de otros *scripts* que realizan las operaciones de sustitución, difusión y permutación descritas en el apartado anterior. Por tanto, para que el cifrador funcione correctamente debe ir acompañado de todo el conjunto de *scripts* en el directorio de MATLAB.
Esta sección se divide en dos partes: la primera para exponer la implementación de la expansión de la clava y la segunda parte tratará del procesamiento de los datos. Con el objetivo de hacer más fácil la lectura, el código se encuentra adjunto en un anexo al final de la memoria, en este apartado se ha redactado un pseudocódigo de cada una de las partes.

4.1. Expansión de la clave

El algoritmo que rige la expansión de la clave, tiene como entrada la clave y como salida las subclaves (*whitening keys* y *round keys*). Para facilitar la transferencia de variables de una parte del algoritmo a otro se almacenan todas las subclaves en dos vectores, uno que contenga todas las *whitening keys* y otro que contenga todas las *round keys*. La función tendrá por tanto una entrada y dos salidas.

Pseudocódigo: Expansión de la clave
function$[wk, rk] = \text{KS80}(r, K)$

Se divide la clave de 80 bits en 5 bloques de 16 bits:
$k_0 = K(1:16)$; $k_1 = K(17:32)$; $k_2 = K(33:48)$;
$k_3 = K(49:64)$; $k_4 = K(65:80)$;

Cada bloque de 16 bits se separa en dos partes de 8 bits
$k_0^L = k_0(1:8)$; $k_0^R = k_0(9:16)$;
$k_1^L = k_1(1:8)$; $k_1^R = k_1(9:16)$;
$k_2^L = k_2(1:8)$; $k_2^R = k_2(9:16)$;
$k_3^L = k_3(1:8)$; $k_3^R = k_3(9:16)$;
$k_4^L = k_4(1:8)$; $k_4^R = k_4(9:16)$;

Cada una de las *round keys* se compone dos de estos bloques:
$wk_0 = [k_0^L, k_1^R]$; $wk_1 = [k_1^L, k_0^R]$;
$wk_2 = [k_4^L, k_3^R]$; $wk_3 = [k_3^L, k_4^R]$;

Se almacenan las 4 *round keys* en un único vector de salida:
$wk = [wk_0, wk_1, wk_2, wk_3]$;

Se crea un vector de ceros para almacenar posteriormente las *round keys*:
$rk = [0..,0]$ ⟵ Debe tener una longitud de $2r * 16$

for $i = 1 : (r-1)$
Se define la constante de ronda:
$(con_{2i}, con_{2i+1}) = (c_{i+1} \mid c_0 \mid c_{i+1} \mid \{00\}_2 \mid c_{i+1} \mid c_0 \mid c_{i+1})$
$\oplus\{\text{0f1e2d3c}\}_{16}$

if $mod(i,5) = 0$ ó if $mod(i,5) = 2$
$(rk_{2i}, rk_{2i+1}) = (con_{2i}, con_{2i+1}) \oplus [k_2, k_3]$
end

if $mod(i,5) = 1$ ó if $mod(i,5) = 4$
$(rk_{2i}, rk_{2i+1}) = (con_{2i}, con_{2i+1}) \oplus [k_0, k_1]$
end

if $mod(i,5) = 3$
$(rk_{2i}, rk_{2i+1}) = (con_{2i}, con_{2i+1}) \oplus [k_4, k_4]$
end

Hay que introducir (rk_{2i}, rk_{2i+1}) dentro de rk:
$rk(32i + 1 : 16(2i + 2)) = (rk_{2i}, rk_{2i+1})$
end

Para convertir números en formato decimal a su secuencia binaria se ha utilizado la función de MATLAB *de2bi*[7] que permite fijar un número de bits para realizar la conversión. La función inversa también se encuentra disponible en MATLAB como *bi2de*. Las funciones *hexToBinaryVector* y su inversa *binaryVectorToHex* se encargan de transformar un valor hexadecimal en un vector binario y viceversa, respectivamente.
El XOR entre vectores binarios se implementa utilizando $bitxor(a, b)$ que, siendo a y b dos vectores binarios, devuelve otro vector resultado de realizar la operación lógica XOR entre a y b bit a bit.
El cálculo de un número a en módulo b se realiza utilizando la función $mod(a, b)$ disponible en MATLAB.

4.2. Procesamiento de los datos

El procesamiento de los datos, a diferencia de la expansión de la clave, requiere de varios archivos para su funcionamiento. Para comenzar, se mostrará el *script* principal en forma de pseudocódigo, que contiene la estructura de procesamiento representada en la Figura 6 [2]. Posteriormente se expondrá el pseudocódigo de las distintas subfunciones que forman parte de la estructura.
La función de encriptado debe tener como entradas los dos vectores que contengan las dos familias de subclaves generados en la expansión de la clave, junto con el bloque de *plaintext*, y debe devolver como salida el bloque de *ciphertext*.

Pseudocódigo: Procesamiento de los datos
function$[Y] = \mathrm{Gr}(X, wk, rk)$
Siendo Y el *ciphertext* y X el *plaintext*

Se divide X en 4 paquetes de 16 bits:
$x_0 = X(1 : 16)$; $x_1 = X(17 : 32)$; $x_2 = X(33 : 48)$; $x_3 = X(49 : 64)$;

Se extraen de wk las 4 subclaves $wk_{i(16)}$ que contiene:
$wk_0 = wk(1 : 16)$; $wk_1 = wk(17 : 32)$;
$wk_2 = wk(33 : 48)$; $wk_3 = wk(49 : 64)$;

[7] Esta función devuelve los bits en orden invertido. Es por eso que en el código esta función va acompañada del uso de *fliplr*, que invierte el orden.

$x_0 = x_0 \oplus wk_0$; $x_2 = x_2 \oplus wk_1$;
for $i = 0 : (r-2)$
    $x_1 = x_1 \oplus F(x_0) \oplus RK(rk, 2i)$;
    $x_3 = x_3 \oplus F(x_2) \oplus RK(rk, 2i+1)$;
    $X = RP([x_0, x_1, x_2, x_3])$;
    $x_0 = X(1:16)$; $x_1 = X(17:32)$; $x_2 = X(33:48)$; $x_3 = X(49:64)$;
end
$x_1 = x_1 \oplus F(x_0) \oplus RK(rk, 2r-2)$;
$x_3 = x_3 \oplus F(x_2) \oplus RK(rk, 2r-1)$;
$x_0 = x_0 \oplus wk_2$; $x_2 = x_2 \oplus wk_3$;
$Y = [x_0, x_1, x_2, x_3]$;

Se puede observar la diferencia en el uso de las *whitening keys* (wk) y las *round keys* (rk): las primeras actúan sobre los datos antes y después del bucle o ronda, mientras que las segundas actúan dentro de cada ronda, de ahí su nombre.
Dentro del algoritmo se utiliza la operación XOR sobre tres argumentos de entrada. Debido a que la función de MATLAB que se está usando para implementar dicha operación (*bitxor*) contiene únicamente dos argumentos de entrada, la operación se realiza aprovechando la propiedad asociativa, de la siguiente forma:

$$a \oplus b \oplus c = \text{bitxor}(a, \text{bitxor}(b, c))$$

También se puede observar que el algoritmo hace uso de diversas funciones, las cuales se describirán a continuación.

Función F (*F*)
Este *script* desempeña el papel de la función F de *Piccolo*, la cual, tal como se detalló en las especificaciones matemáticas, tiene como argumento de entrada y salida un bloque de 16 bits. El pseudocódigo que se presenta a continuación contiene la estructura de F, la cual hace uso de otros dos archivos que se corresponden con la matriz de difusión y la S-*box*.

Pseudocódigo: Función F
function$[Y]$ = F(X)
X e Y son dos vectores de 16 bits.

Se divide X en 4 paquetes de 4 bits:
$x_0 = X(1:4)$; $x_1 = X(5:8)$; $x_2 = X(9:12)$; $x_3 = X(13:16)$;

Se declara cada x_i en representación hexadecimal:
$x_0 = \{x_0\}_{16}$; $x_1 = \{x_1\}_{16}$; $x_2 = \{x_2\}_{16}$; $x_3 = \{x_3\}_{16}$;

A cada variable hexadecimal se le aplica la S-*box*:
$x_0 = S(x_0)$; $x_1 = S(x_1)$; $x_2 = S(x_2)$; $x_3 = S(x_3)$;

Se devuelve cada valor de salida a representación binaria y se agrupa en un vector de 16 bits:
$x_0 = \{x_0\}_2$; $x_1 = \{x_1\}_2$; $x_2 = \{x_2\}_2$; $x_3 = \{x_3\}_2$;
$X = [x_0, x_1, x_2, x_3]$;

A este vector de 16 bits se le aplica la matriz de difusión:
$X = M(X)$;

Por último se vuelve a aplicar la S-*box* siguiendo el procedimiento anterior:
$x_0 = X(1:4)$; $x_1 = X(5:8)$; $x_2 = X(9:12)$; $x_3 = X(13:16)$;
$x_0 = \{x_0\}_{16}$; $x_1 = \{x_1\}_{16}$; $x_2 = \{x_2\}_{16}$; $x_3 = \{x_3\}_{16}$;
$x_0 = S(x_0)$; $x_1 = S(x_1)$; $x_2 = S(x_2)$; $x_3 = S(x_3)$;
$x_0 = \{x_0\}_2$; $x_1 = \{x_1\}_2$; $x_2 = \{x_2\}_2$; $x_3 = \{x_3\}_2$;

La concatenación de estos cuatro vectores formaría el vector de salida Y:
$Y = [x_0, x_1, x_2, x_3]$;

Para que F funcione adecuadamente, son necesarios los archivos que contienen la matriz de difusión (M) y la S-*box* (S).

S-*box* (S)
El *script* que contiene la función S está formado por una secuencia de 16 condicionales *if*, de forma que a cada dato hexadecimal de entrada se le asigne su valor de salida según el Cuadro 1. Dicha estructura es poco significativa para el entendimiento del funcionamiento de *Piccolo* en MATLAB y mostrarla en forma de pseudocódigo ensuciaría la memoria innecesariamente, de manera que simplemente se adjunta el código en el anexo.
Cabe, sin embargo, reseñar que para utilizar los condicionales en MATLAB ha sido necesario leer las variables en formato decimal en lugar de hexadecimal, devolviendo posteriormente el formato de entrada.

Matriz de difusión (M)
La operación de difusión consiste en aplicar la llamada matriz de difusión (M), que es una matriz cuadrada de orden 4, a un vector binario de 16 bits, el cual se divide en 4 paquetes de 4 bits.

Pseudocódigo: Matriz de difusión
function$[Y]$ = M(X)
X e Y son dos vectores de 16 bits.

Se divide la entrada en 4 paquetes de 4 bits:
$x_0 = X(1:4)$; $x_1 = X(5:8)$; $x_2 = X(9:12)$; $x_3 = X(13:16)$;

Se pasa cada paquete a formato decimal:

$x_0 = \{x_0\}_{10}$; $x_1 = \{x_1\}_{10}$; $x_2 = \{x_2\}_{10}$; $x_3 = \{x_3\}_{10}$;

Se aplica la matriz operando filas con columnas[8]:
$y_0 = \text{gf2e4}(2, x_0) \oplus \text{gf2e4}(3, x_1) \oplus \text{gf2e4}(1, x_2) \oplus \text{gf2de}(1, x_3)$;
$y_1 = \text{gf2e4}(1, x_0) \oplus \text{gf2e4}(2, x_1) \oplus \text{gf2e4}(3, x_2) \oplus \text{gf2de}(1, x_3)$;
$y_2 = \text{gf2e4}(1, x_0) \oplus \text{gf2e4}(1, x_1) \oplus \text{gf2e4}(2, x_2) \oplus \text{gf2de}(3, x_3)$;
$y_3 = \text{gf2e4}(3, x_0) \oplus \text{gf2e4}(1, x_1) \oplus \text{gf2e4}(1, x_2) \oplus \text{gf2de}(2, x_3)$;

Los valores decimales se devuelven a formato binario:
$x_0 = \{x_0\}_2$; $x_1 = \{x_1\}_2$; $x_2 = \{x_2\}_2$; $x_3 = \{x_3\}_2$;

El vector de salida es la concatenación de los 4 vectores x_i:
$Y = [x_0, x_1, x_2, x_3]$;

La función *gf2e4* se encarga de multiplicar dos números decimales (a,b) dentro del campo $GF(2^4)$ utilizando el polinomio irreducible x^4+x+1 devolviendo c. Su estructura es sencilla y se apoya en funciones existentes en MATLAB. Se ha obtenido de la referencia [4]. Su código puede consultarse en el anexo.

$$\text{Si} \quad (a,b) \in (0,15)_{10} \quad \text{y} \quad c = \text{gf2e4}(a,b) \quad \Rightarrow \quad c \in (0,15)_{10}$$

Función *Round Permutation* (*RP*)
Esta función realiza la operación de permutación de cada ronda, de ahí su nombre. Divide la entrada, de 64 bits, en 8 paquetes de 8 bits, los permuta según las especificaciones, y los reagrupa en otro vector de salida de 64 bits.

Pseudocódigo: Permutación de ronda
function$[Y] = \text{RP}(X)$
X e Y son dos vectores de 64 bits.

Se divide la entrada en 8 paquetes de 8 bits:
$x_0 = X(1:8)$; $x_1 = X(9:16)$; $x_2 = X(17:24)$; $x_3 = X(25:32)$;
$x_4 = X(33:40)$; $x_5 = X(41:48)$; $x_6 = X(49:56)$; $x_7 = X(57:64)$;

Se reordenan los paquetes según las especificaciones de RP:
$Y = [x_2, x_7, x_4, x_1, x_6, x_3, x_0, x_5]$;

Función *RK*
Al no conocer a priori la longitud del vector que contiene las *round keys* (rk), ya que depende del número de rondas que se ejecute el algoritmo (en su versión estándar serían 25), la extracción de una subclave rk_i del vector global rk se realiza mediante esta función. Tiene como entradas el vector rk y el índice de la subclave que se desea extraer i, y como salida la propia subclave rk_i.

[8] La multiplicación de la matriz por el vector utilizando operaciones XOR se ha consultado en la referencia [4].

Pseudocódigo: Función RK
function$[rk_i] = \mathrm{RK}(rk, i)$
rk_i es un vector de 16 bits.
rk contiene $2r * 16$ bits e i es un número entero.

Se extraen de rk los 16 bits que corresponden a rk_i:
$rk_i = rk(16i + 1 : 16(i + 1))$;

Se trata de una función bastante sencilla, cuya única dificultad es determinar los valores $16i + 1$ y $16(i + 1)$ que delimitan cada rk_i dado el índice i.

4.3. Ensamblaje de las partes del cifrador

Para terminar de implementar el algoritmo, únicamente falta unir en un mismo archivo sus dos partes principales: la expansión de la clave seguida del procesamiento de los datos, según la Figura 3. De manera que el *script* final, tenga como entrada la clave y el bloque de *plaintext*, y como salida el bloque de *ciphertext*. También se ha dejado como parámetro ajustable el número de rondas r que se ejecuta el algoritmo, ya que puede resultar útil en posteriores chequeos de funcionalidad de cada parte.

Pseudocódigo: Piccolo-80
function$[Ciphertext] = \mathrm{Piccolo80}(Plaintext, Key, r)$
Ciphertext: 64 bits; *Plaintext*: 64 bits; Key: 80 bits; r: número entero;

Parte 1: Expansión de la clave:
$[wk, rk] = \mathrm{KS80}(r, Key)$;

Parte 2: Procesamiento de los datos:
$Ciphertext = \mathrm{Gr}(Plaintext, wk, rk, r)$;

Como se puede observar, el *script* final de *Piccolo* tiene una estructura bastante sencilla, únicamente consiste en colocar las dos partes de forma sucesiva según la Figura 3. Con este archivo ya se tendría implementado completamente el cifrador, a falta del chequeo de funcionalidad.
En el pseudocódigo aparecen las entradas y salidas en formato binario, ya que de esta forma trabajará la aplicación. Sin embargo se puede hacer que el cifrador trabaje en formato hexadecimal fácilmente, introduciendo en el *script* dos líneas de código que cambien el formato de las entradas a binario, y una a la salida que devuelva el formato a hexadecimal.

key	00112233 44556677 8899
plaintext	01234567 89abcdef
ciphertext	8d2bff99 35f84056

Cuadro 2: Vectores de test de *Piccolo-80* obtenidos de [2].

4.4. Prueba de funcionalidad del *software*

Una vez realizado el traslado completo del cifrador al entorno de MATLAB, es necesario verificar la correcta implementación de todas las las partes del algoritmo. Lo normal en estas situaciones es que existan fallos en la escritura, ya sean despistes a la hora de redactar alguna orden o errores de índole matemática.

Ejecución en la ventana de comandos
Para comenzar se comprueba que todas las partes se encuentran bien ensambladas y que el cifrador se ejecuta sin problemas de dimensionalidad de los vectores. Esta prueba se ha ido realizando con cada uno de los archivos individualmente. Una vez se confirma que el algortimo se ejecuta, tomando un bloque de 64 bits y devolviendo otro, se confirma que todas las estructuras implementadas son coherentes con las longitudes de los vectores y que las funciones de MATLAB utilizadas admiten las variables de entrada y salida que se les han asignado.

Verificación mediante vectores test
Sin embargo, con esta prueba no basta para confirmar que el cifrador que se ha implementado se corresponde con *Piccolo-80*. Para ello, es necesario testear manualmente el algoritmo, utilizando vectores de test conocidos, y comprobar que la salida de nuestro algoritmo en MATLAB es la especificada por los vectores de test. Afortunadamente, el artículo [2] dispone de vectores de test para realizar la prueba (Cuadro 2). Al aparecer en formato hexadecimal, se retoca el archivo de *Piccolo-80* para que trabaje en este formato.

Si los vectores de test coinciden con la entrada y salida del algoritmo en MATLAB, se tendría ya implementado el cifrador perfectamente funcional.

Depuración del código manualmente
En este caso, los vectores test no coincidieron en un primer momento con la entrada y salida del cifrador, en consecuencia, hubo que realizar una prueba adicional para encontrar el error o errores: depurar algoritmo.
Esta prueba consiste en estudiar cada parte del cifrador por separado, e introduciendo entradas sencillas a cada función, calcular manualmente la salida que se debería obtener, para verificar que la función matemática que se desea implementar coincide con la función que está realizando el código. En las funciones que contienen bucles, se ajusta el número de rondas al mínimo posible[9], para facilitar los cálculos.

[9] Por esta razón se introdujo el número de rondas r como parámetro ajustable.

Una vez chequeada la funcionalidad de cada una de las partes mediante este método, se solventaron los errores encontrados y se volvió a realizar la prueba de los vectores de test con éxito. En este punto, se puede decir que el cifrador está correctamente implementado en MATLAB.

5. Desarrollo de la aplicación de testeo

El desarrollo de la aplicación tiene como objetivo la generación automática de vectores de test de un cifrador, como puede ser *Piccolo-80*. El objetivo es crear un *script* de MATLAB que sea capaz de generar tantos vectores de test como se desee a partir de un algoritmo criptográfico previamente implementado, y guardar los resultados para su posterior uso.
Como se ha comentado brevemente en la introducción, el ámbito de utilidad de esta aplicación es el chequeo de funcionalidad de algoritmos implementados en *hardware*. Una vez diseñado el *hardware* de un cifrador, es necesario comprobar que funcione correctamente, para ello se estudia su respuesta a vectores de test. La función que desempeña esta aplicación en este proceso es la de generar esos vectores de test. Teniendo el cifrador completamente implementado en MATLAB, como se tiene *Piccolo*, una vez se haya verificado que funciona de forma correcta mediante las pruebas realizadas en el apartado 4.4., se pueden generar todos los vectores test que se deseen con la certeza de que son completamente fiables. Con la aplicación, se podrían generar vectores de test de manera automática. Por ejemplo, se podría obtener en cuestión de segundos un archivo con 100 vectores de test. Una vez se recogen esos datos, pueden ser utilizados por un diseñador para chequear la funcionalidad de su dispositivo en *hardware*. El diseñador puede alimentar su dispositivo con los datos de entrada proporcionados por la aplicación, recopilar las respuestas que obtiene, y compararlas con los datos de salida de la aplicación. De esta forma se puede verificar el correcto funcionamiento de un diseño antes de su fabricación.
La idea es que la aplicación sea capaz de configurarse para generar vectores de test de cualquier cifrador por bloques instalado en MATLAB. Llevará incorporado el algoritmo *Piccolo-80* pero se añade una opción para que el usuario pueda añadir fácilmente cualquier otro cifrador, idependientemente de la longitud del bloque o la clave.

5.1. Pseudocódigo de la aplicación

La aplicación alimentará el algoritmo del cifrador sucesivas veces mediante un bucle, con datos de entrada (*plaintext* y clave) generados pseudoaleatoriamente en cada ronda del bucle. En cada ronda se generará el *ciphertext* correspondiente. Antes de terminar el bucle se guardarán los datos de entrada y salida

antes de proceder a la generación del siguiente vector de test.
A continuación se muestra el pseudocódigo de la aplicación y posteriormente se justificará su estructura detenidamente:

Pseudocódigo: Aplicación vectores de test
Introducir la opción (1, 2 o 3): opt =?

if $opt = 1 \longrightarrow$ Se testea *Piccolo-80*
Introducir el número de vectores de test: n =?

Se generan tres matrices de ceros que almacenarán los datos:
$Plaintext = ceros(n \times 64)$;
$Key = ceros(n \times 80)$;
$Ciphertext = ceros(n \times 64)$;

for $i = 1 : n$
$Plaintext(i \times 64) = \text{random}[..., 0, 1, ...](1 \times 64)$;
$Key(i \times 80) = \text{random}[..., 0, 1, ...](1 \times 80)$;
$Ciphertext(i \times 64) = \text{Piccolo80}(Plaintext(i \times 64), Key(i \times 80))$;
end

Se guardan los datos de entrada y salida en tres archivos:
Guardar $Plaintext(i \times 64) \longrightarrow$ Plaintext.mat
Guardar $Key(i \times 80) \longrightarrow$ Key.mat
Guardar $Ciphertext(i \times 64) \longrightarrow$ Ciphertext.mat
end

if $opt = 2 \longrightarrow$ Se abre un *script* para introducir un nuevo algoritmo.
En el nuevo *script* se debe introducir el nombre del nuevo algoritmo donde se indica. Se guarda el *script* y se vuelve a la ventana de comandos.
end.

if $opt = 3 \longrightarrow$ Se testea el algoritmo introducido en la opción 2.
Introducir el número de vectores de test: n =?
Introducir la longitud del bloque *plaintext/ciphertext* en bits: $long$ =?
Introducir la longitud de la clave en bits: $clave$ =?

Se generan tres matrices de ceros que almacenarán los datos:
$Plaintext = ceros(n \times long)$;
$Key = ceros(n \times clave)$;
$Ciphertext = ceros(n \times long)$;

for $i = 1 : n$
$Plaintext(i \times long) = \text{random}[..., 0, 1, ...](1 \times long)$;
$Key(i \times clave) = \text{random}[..., 0, 1, ...](1 \times clave)$;

$Ciphertext(i \times long) = \text{AlgoritmoNuevo}(Plaintext(i \times long), Key(i \times clave))$;

end

Se guardan los datos de entrada y salida en tres archivos:
Guardar $Plaintext(i \times long) \longrightarrow$ Plaintext.mat
Guardar $Key(i \times clave) \longrightarrow$ Key.mat
Guardar $Ciphertext(i \times long) \longrightarrow$ Ciphertext.mat

end

5.2. Funcionamiento

El pseudocódigo muestra que la esencia de la aplicación es un *script* que, al ejecutarse, permite al usuario elegir una de las 3 opciones que son:

- **Opción 1**: Al pulsar 1, la aplicación procede a testear *Piccolo*, pregunta al usuario cuántos vectores de test desea generar (n), y mediante un bucle *for* comienza a generar los datos de entrada de forma pseudoaleatoria y a registrar la salida de cada ronda. Este proceso se realiza creando antes del bucle tres matrices de ceros que contendrán cada uno de los datos. Luego, cada ronda del bucle rellena una de las filas de las matrices con el *plaintext*, clave y *ciphertext* de cada ronda, para $i = 1$ rellena la primera fila de cada matriz y para $i = 37$ la fila 37, así hasta llegar a la fila n-ésima. El número de columnas de cada matriz deben ser las longitudes de los bloques de *plaintext*/*ciphertext* y de la clave. Una vez terminado el bucle, se guardan los datos que han recogido las matrices en tres archivos, uno que contiene los datos de *plaintext*, otro los de la clave y otro los del *ciphertext*.

- **Opción 2**: Cuando se pulsa la opción 2, la aplicación abre automáticamente un *script* donde se debe introducir el nombre del algoritmo nuevo que se desea introducir. Para que se ejecute correctamente, el nuevo algoritmo debe tener forma de cifrador por bloques, es decir, dos entradas: *plaintext* y clave en forma de vectores binarios y como salida: el *ciphertext*:

  $$Ciphertext = \text{NuevoAlgoritmo}(Plaintext, Key)$$

  Si el algoritmo no tiene esta forma, es preciso adaptarlo para que la aplicación pueda testearlo. Por ejemplo, en la versión de *Piccolo* que se ha implementado, se tenía como entra el parámetro r, dicha variable debería retirarse de la casilla de argumentos de entrada e introducir su valor dentro del código del agoritmo.

- **Opción 3**: Esta opción solo debe ser ejecutada si se ha introducido previamente un algoritmo mediante la opción 2. El procedimiento para generar

los vectores de test es idéntico al de la opción 1, con la diferencia de que deja como parámetros ajustables las longitudes del bloque *plaintext/ciphertext* (*long*) y de la clave (*clave*), para que el usuario pueda ajustar el proceso a las especificaciones de su algoritmo.

La aplicación, por tanto, se ha diseñado sobre las especificaciones del algoritmo implementado en este trabajo: *Piccolo-80*, pero se ha ampliado su rango de funcionalidad de forma que pueda generar vectores de test de cualquier cifrador por bloques. Únicamente es necesario haber implementado previamente dicho cifrador en MATLAB y colocar sus archivos en el directorio, para que la aplicación pueda acceder a él.
Se ha mencionado que al ejecutar la opción 2 se abre un *script* auxiliar en el que se debe introducir el nombre del nuevo algoritmo. Dicho *script* tiene la siguiente forma:

Pseudocódigo: Algoritmo nuevo
function$[Ciphertext]$ = AlgoritmoNuevo$(Plaintext, Key)$

$Ciphertext$ = newALG$(Plaintext, Key)$
Se debe sustituir el nombre *newALG* por el del *script* que contiene al nuevo algoritmo.

Si nos fijamos en el pseudocódigo de la aplicación, en la opción 3 se da la orden de testear un algoritmo llamado *AlgoritmoNuevo*, al introducir en la casilla *newALG* el nombre de nuestro algoritmo, cada vez que la aplicación invoque la función *AlgoritmoNuevo* estará realmente invocando nuestro nuevo algoritmo, gracias a este *script* auxiliar.

5.3. Prueba de ejecución

Con el objetivo de comprobar el correcto funcionamiento de la aplicación, se va a realizar una prueba de cada una de las opciones que admite la aplicación y se verificará que se obtienen los datos correctamente.

Opción 1: Generación de vectores de test de *Piccolo-80*
Se ejecutará la aplicación en su opción 1 (*Piccolo-80*) y se comprobará que se obtienen los tres ficheros con los datos de *plaintext*, clave y *ciphertext*. Se adjunta una imagen donde se muestra la interfaz de la aplicación desde la ventana de comandos de MATLAB (Figura 9) [10].
El parámetro de testeo, que en este caso es únicamente el número de vectores test a generar, se ha fijado en 5. La ejecución ha sido satisfactoria, puesto que se han obtenido todos los datos perfectamente, se encuentran disponibles en el directorio de MATLAB y se muestran en el Cuadro 3 en forma hexadecimal.

[10] Las Figuras de la ejecución de la aplicación se encuentran en el Anexo II.

Plaintext	*Key*	*Ciphertext*
D9DA7BEA1A31D8AB	E2A27B4E855C5C5C50ED	DC5E721C7A1CBC51
00C48388EA9B0FB7	C204C2C12D3997157A6F	CBBD7C822579F02D
C8E4BBE432C40D35	F2716092EBA02E379817	5342BB1C3682D4CD
D636A144551DF49A	DE37F01F2E724AC0AB35	7AE2E28987216B44
BE3A20FF7A7D7FCA	D005A3321BBF085C2BC6	8D0579EC84EF6FED

Cuadro 3: Resultados de los 5 vectores de test generados mediante la aplicación. Ejecución en la Figura 9.

Opción 2: Configuración de un nuevo algoritmo
Durante esta opción se abre automáticamente un fichero en el que el usuario debe introducir el nombre del algoritmo que desea testear (Figura 10). La casilla *newALG* debe ser sustituida por el nombre del fichero que contenga el nuevo algoritmo.
Una vez introducido el nombre, se guardan los cambios pulsando *SAVE*, se cierra la ventana del *script* que se ha abierto (no MATLAB), y se vuelve a la ventana de comandos, donde el usuario se encuentra un mensaje para que vuelva a ejecutar la aplicación, en este caso en su opción 3 (Figura 11).

Opción 3: Generación de vectores de test del nuevo algoritmo
Una vez introducido el nombre del nuevo algoritmo mediante la opción 2, no antes, se pasa a testearlo, para eso se pulsa la opción 3. Deben introducirse más parámetros, debido a que la aplicación debe recavar información básica sobre las especificaciones del nuevo algoritmo.
Se ha realizado la prueba de generar 4 vectores de test para la versión de *Piccolo* de 128 bits de clave [11] (Figura 12).
Se ha programado la prueba para que devuelva 4 vectores de test y se han ajustado los parámetros del nuevo algoritmo: bloque de 64 bits y clave de 128 bits. La prueba se ha realizado satisfactoriamente, se obtienen los datos de forma correcta.

Prueba de rendimiento con *Piccolo-80*
Para comprobar la velocidad de computo de la aplicación para generar vectores de test, de un algoritmo como *Piccolo*, se mide el tiempo que tarda MATLAB en ejecutar la aplicación. Se realizará la prueba para la generación de 100 vectores. Para medir tiempos en MATLAB se utiliza la herramienta *tic-toc*, la cual mide el tiempo que transcurre desde que se ejecuta la orden de inicio (*tic*), hasta que se ejecuta la orden de finalización de la cuenta (*toc*). Colocando la orden *tic* dentro de nuestro código justo antes del bucle *for* que genera los vectores de test, y la orden *toc* al finalizar dicho bucle, MATLAB devolverá el tiempo transcurrido.

[11] Dicha versión se intentó implementar, sin embargo, no superó la prueba de verificación con los vectores de test de [2]. Aún así en este apartado nos sirve como algoritmo cualquiera.

Se ha realizado la prueba con la generación de 100 vectores sobre *Piccolo-80* y el resultado ha sido de 189,52 segundos (Figura 13).

5.4. El entorno de ataques del IMSE

En el Instituto de Microelectrónica de Sevilla existe una línea de investigación dedicada a la microelectrónica para la seguridad. Dentro de este contexto, los investigadores de esta línea de estudios, han establecido un entorno de ataque lateral sobre cifradores, donde esta aplicación jugaría un papel fundamental (véase la Figura 8).
La aplicación desarrollada en este trabajo sería la encargada de generar los tres archivos de *Plaintext*, *Key* y *Ciphertext* que posteriormente se usarán para testear el cifrador (*hardware*) a través de un sistema de control *FPGA* (*Field Programmable Gate Array*). Para chequear el funcionamiento del *hardware* se comparan los datos de *Ciphertext* obtenidos directamente de la aplicación y aquellos obtenidos tras testear el cifrador (*hardware*).
A parte, el *hardware* está monitorizado mediante un osciloscopio que mide trazas EM o de consumo. Todo el trasvase de datos se controla desde el propio programa MATLAB.

6. Conclusiones

Para terminar, es conveniente hacer un pequeño balance sobre los objetivos iniciales de este trabajo y los resultados obtenidos. En principio se plantearon dos objetivos fundamentales:

- La implementación del algoritmo criptográfico *Piccolo* en el entorno de MATLAB.
- El desarrollo en MATLAB de una aplicación capaz de generar vectores de test de algortimos criptográficos.

La implementación de *Piccolo-80* ha sido realizada con éxito, y certificada mediante varias pruebas, siendo definitiva la prueba realizada con los vectores de test adjuntos en [2]. En este trabajo se ha propuesto una implementación de dicho algoritmo en MATLAB que puede ser útil en el futuro para testear posibles dispositivos que hagan uso de este cifrador. Es más, de la mano de la aplicación, cualquiera podría generar todos los vectores de test que considerara necesarios para chequear el correcto funcionamiento de su dispositivo en cuestión de segundos.
En cuanto a la aplicación, a parte de generar vectores de test de *Piccolo-80*, se ha programado para que de forma sencilla se pueda utilizar sobre cualquier algoritmo de encriptado, una vez este haya sido implementado en MATLAB. Incluso, podría utilizarse sobre algoritmos de desencriptado, pero siendo conscientes de que el archivo *plaintext* realmente contiene al *ciphertext* y viceversa. En principio, cualquier algoritmo criptográfico de clave simétrica podría ser objeto de

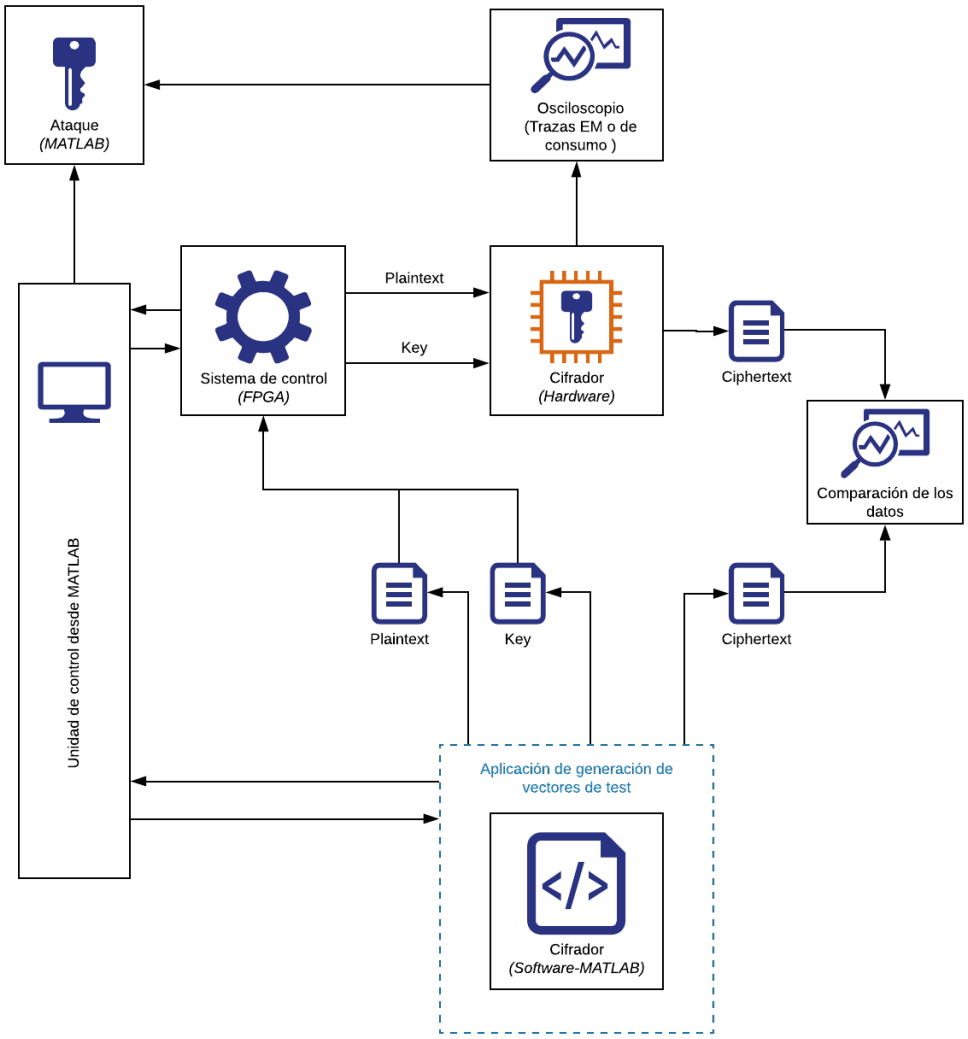

Figura 8: Entorno de ataques del IMSE.

uso de esta aplicación, la cual es de funcionamiento sencillo pero bastante útil. Destaca su utilidad en entornos en los que se trabaje diseñando dispositivos electrónicos destinados a la seguridad, como ocurre en el IMSE (Instituto de Microelectrónica de Sevilla), donde existe una línea de investigación entera dedicada a el diseño de dispositivos microelectrónicos para la seguridad.

Referencias

[1] Stallings, W., *Cryptography and Network Security: Principles and Practice*, 2005. Prentice Hall.

[2] Shibutani, K. and Isobe, T. and Hiwatari, H. and Mitsuda, A. and Akishita, T. and Shiari, T., *Piccolo: An Ultra-Lightweight Blockcipher*, 2011. Sony Corporation.

[3] Salomon, D. *Block Ciphers*. In: *Coding for Data and Computer Comunications*, 2005. Springer, Boston, MA.

[4] Wolkerstorfer, J. and Oswald, E. and Lamberger, M., *An ASIC Implementation of the AES SBoxes*, 2002. Springer-Verlag, Berlín.

[5] Ashur, T. and Dunkelman, U. and Masalha, N., *Linear Cryptanalysis Reduced Round of Piccolo-80*, Department of Electrical Engineering, ESAT/COSIC, KU Leuven, Leuven, Bélgica. Springer Nature Switzerland AG (2019).

[6] Biham, E., Biryukov, A., Shamir, A.: *Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials*. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 12–23. Springer, Heidelberg (1999).

[7] Bogdanov, A., Khovratovich, D., Rechberger, C.: *Biclique cryptanalysis of the Full AES*. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 344–371. Springer, Heidelberg (2011).

[8] Bogdanov, A., Rechberger, C.: *A 3-subset meet-in-the-middle attack: cryptanalysis of the lightweight block cipher KTANTAN*. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 229–240. Springer, Heidelberg (2011).

[9] Matsui, M.: *Linear cryptanalysis method for DES cipher*. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994).

[10] Matsui, M.: *The first experimental cryptanalysis of the data encryption standard*. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 1–11. Springer, Heidelberg (1994).

[11] Hatzivasilis, G. and Floros, G. and Papaefstathion, I. and Manifavas, C.: *Lightweight password hashing scheme for embedded systems.* In: 9th WG 11.2 International Conference on Information Security Theory and Practice (WISTP), IFIP, Springer, LNCS, 9311, pp. 249-259 (2015).

Anexo I: Códigos de *Piccolo-80*

Función *KS80*: expansión de la clave

```
function [wk,rk] = KS80(r,K)

%We first of all divide K into 5 16-bit ki
k0 = K(1:16);  k1 = K(17:32);  k2 = K(33:48);  k3 = K
    (49:64);  k4 = K(65:80);

%Now we separate the left and the right part of each ki
k0L = k0(1:8); k0R = k0(9:16);
k1L = k1(1:8); k1R = k1(9:16);
k3L = k3(1:8); k3R = k3(9:16);
k4L = k4(1:8); k4R = k4(9:16);

%We now can get wk:
wk0 = [k0L,k1R]; wk1 = [k1L,k0R]; wk2 = [k4L,k3R]; wk3 =
    [k3L,k4R];
wk = [wk0,wk1,wk2,wk3];

%Now we need the round key rk:
rk = zeros(16*2*r,1);
%Because the idea is to fill this zeros with 2r vectors
    of length 16-bits

for i = 0:(r-1)
     %We have to define the constant values of each
         iteration.
     con1 = [fliplr(de2bi(i+1,5)),fliplr(de2bi(0,5)),
         fliplr(de2bi(i+1,5)),fliplr(de2bi(0,2)),fliplr(
         de2bi(i+1,5)),fliplr(de2bi(0,5)),fliplr(de2bi(i
         +1,5))];
     con2 = hexToBinaryVector('0f1e2d3c',32);
     con2i2i1 = bitxor(con1,con2);
     if mod(i,5) == 0
          rk2i2i1 = bitxor(con2i2i1,[k2,k3]);
          %rk2i2i1 is the concatenation of rk(2i) and rk(2
              i+1).
     end
     if mod(i,5) == 2
          rk2i2i1 = bitxor(con2i2i1,[k2,k3]);
     end
     if mod(i,5) == 1
          rk2i2i1 = bitxor(con2i2i1,[k0,k1]);
     end
```

```
35      if mod(i,5) == 4
36          rk2i2i1 = bitxor(con2i2i1,[k0,k1]);
37      end
38      if mod(i,5) == 3
39          rk2i2i1 = bitxor(con2i2i1,[k4,k4]);
40      end
41
42      rk(2*i*16 + 1:(2*i+2)*16) = rk2i2i1;
43  end
44  rk = rk';
```

Función *Gr*: procesamiento de los datos

```
1   function [Y] = Gr_encrypt(X,wk,rk,r)
2   %Gr converts a 64-bit data X in another 64-bit data Y (
        encrypts X in Y),
3   %using four 16-bit whitening keys and 2r 16-bit round
        keys (r rounds).
4   %Notice that the four 16-bit wkeys are introduced as a
        64-bit vector.
5   x0 = X(1:16);  x1 = X(17:32);  x2 = X(33:48);  x3 = X
        (49:64);
6   wk0 = wk(1:16);  wk1 = wk(17:32);  wk2 = wk(33:48);  wk3
        = wk(49:64);
7
8   %Now let's make the whitening keys act on the data X:
9   x0 = bitxor(x0,wk0); x2 = bitxor(x2,wk1);
10
11  %Now let's make the round keys act:
12  for i = 0:(r-2)
13      x1 = bitxor(x1,bitxor(F(x0),RK(rk,2*i)));
14      x3 = bitxor(x3,bitxor(F(x2),RK(rk,2*i+1)));
15      X_RP = RP([x0,x1,x2,x3]);
16      x0 = X_RP(1:16);  x1 = X_RP(17:32);
17      x2 = X_RP(33:48);  x3 = X_RP(49:64);
18  end
19  x1 = bitxor(x1,bitxor(F(x0),RK(rk,2*r-2)));
20  x3 = bitxor(x3,bitxor(F(x2),RK(rk,2*r-1)));
21  x0 = bitxor(x0,wk2);  x2 = bitxor(x2,wk3);
22
23  %Now we build Y with the resultant values of X0, X1, X2,
        X3:
24  Y = [x0,x1,x2,x3];
25  end
```

Función F

```
function [Y] = F(X)
% This is the F-function, which takes a 16-bit binary vector and convert it
% into another 16-bit binary vector. It is used in the data processing part
% of PICCOLO.

x0 = X(1:4);  x1 = X(5:8);  x2 = X(9:12);  x3 = X(13:16);

%We convert every 4-bit string XFi into hexadecimal values:
hex0 = binaryVectorToHex(x0);
hex1 = binaryVectorToHex(x1);
hex2 = binaryVectorToHex(x2);
hex3 = binaryVectorToHex(x3);

%Now we apply the S-box in hexadecimal form:
hex0 = Sdec(hex0);
hex1 = Sdec(hex1);
hex2 = Sdec(hex2);
hex3 = Sdec(hex3);

%And we reconvert it to binary strings:
x0 = hexToBinaryVector(hex0,4);
x1 = hexToBinaryVector(hex1,4);
x2 = hexToBinaryVector(hex2,4);
x3 = hexToBinaryVector(hex3,4);

X = [x0,x1,x2,x3];

%We apply the diffusion matrix:

X = diffusionMatrix(X);

%We have to apply the S-box again:
x0 = X(1:4);  x1 = X(5:8);  x2 = X(9:12);  x3 = X(13:16);

%We convert every 4-bit string XFi into hexadecimal values:
hex0 = binaryVectorToHex(x0);
hex1 = binaryVectorToHex(x1);
hex2 = binaryVectorToHex(x2);
hex3 = binaryVectorToHex(x3);

%Now we apply the S-box in hexadecimal form:
```

```
hex0 = Sdec(hex0);
hex1 = Sdec(hex1);
hex2 = Sdec(hex2);
hex3 = Sdec(hex3);

%And we reconvert it to binary strings:
x0 = hexToBinaryVector(hex0,4);
x1 = hexToBinaryVector(hex1,4);
x2 = hexToBinaryVector(hex2,4);
x3 = hexToBinaryVector(hex3,4);

Y = [x0,x1,x2,x3];

end
```

Matriz de difusión

```
function [Y] = diffusionMatrix(X)
%This function is used in Piccolo to apply the diffusion
    matrix to a
%16-bit binary vector.

x0 = X(1:4); x1 = X(5:8); x2 = X(9:12); x3 = X(13:16);
dx0 = bi2de(fliplr(x0)); dx1 = bi2de(fliplr(x1)); dx2 =
    bi2de(fliplr(x2)); dx3 = bi2de(fliplr(x3));

dy0 = bitxor(gf2e4(2,dx0),bitxor(gf2e4(3,dx1),bitxor(
    gf2e4(1,dx2),gf2e4(1,dx3))));
dy1 = bitxor(gf2e4(1,dx0),bitxor(gf2e4(2,dx1),bitxor(
    gf2e4(3,dx2),gf2e4(1,dx3))));
dy2 = bitxor(gf2e4(1,dx0),bitxor(gf2e4(1,dx1),bitxor(
    gf2e4(2,dx2),gf2e4(3,dx3))));
dy3 = bitxor(gf2e4(3,dx0),bitxor(gf2e4(1,dx1),bitxor(
    gf2e4(1,dx2),gf2e4(2,dx3))));

y0 = fliplr(de2bi(dy0,4)); y1 = fliplr(de2bi(dy1,4)); y2
    = fliplr(de2bi(dy2,4)); y3 = fliplr(de2bi(dy3,4));

Y = [y0,y1,y2,y3];

end
```

Multiplicación en $GF(2^4)$

```
function [q_dec] = gf2e4(a_dec,b_dec)
%This function takes two 4-bit numbers and multiply them
    over GF(2^4)
```

```
3  % Reference paper: "An ASIC Implementation of the AES
       SBoxes".
4  a = fliplr(de2bi(a_dec,4)); b = fliplr(de2bi(b_dec,4));
5
6  a0 = a(4); a1 = a(3); a2 = a(2); a3 = a(1);
7  b0 = b(4); b1 = b(3); b2 = b(2); b3 = b(1);
8  aA = xor(a0,a3); aB = xor(a2,a3);
9
10 q0 = xor(and(a0,b0),xor(and(a3,b1),xor(and(a2,b2),and(a1,
       b3))));
11 q1 = xor(and(a1,b0),xor(and(aA,b1),xor(and(aB,b2),and(xor
       (a1,a2),b3))));
12 q2 = xor(and(a2,b0),xor(and(a1,b1),xor(and(aA,b2),and(aB,
       b3))));
13 q3 = xor(and(a3,b0),xor(and(a2,b1),xor(and(a1,b2),and(aA,
       b3))));
14
15 q = [q3,q2,q1,q0];
16 q_dec = bi2de(fliplr(q));
17
18 end
```

Función S-*box*

```
1  function [hex_out] = Sdec(hex_in)
2  % This function converts a hexadecimal value into another
       following a table
3  % given in the specifications of PICCOLO.
4
5  x = hex2dec(hex_in);
6
7  if x == 0;
8      y = 14;
9  end
10
11 if x == 1;
12     y = 4;
13 end
14
15 if x == 2;
16     y = 11;
17 end
18
19 if x == 3;
20     y = 2;
21 end
```

```
22
23 if x == 4;
24     y = 3;
25 end
26
27 if x == 5;
28     y = 8;
29 end
30
31 if x == 6;
32     y = 0;
33 end
34
35 if x == 7;
36     y = 9;
37 end
38
39 if x == 8;
40     y = 1;
41 end
42
43 if x == 9;
44     y = 10;
45 end
46
47 if x == 10;
48     y = 7;
49 end
50
51 if x == 11;
52     y = 15;
53 end
54
55 if x == 12;
56     y = 6;
57 end
58
59 if x == 13;
60     y = 12;
61 end
62
63 if x == 14;
64     y = 5;
65 end
66
67 if x == 15;
```

```
68      y = 13;
69  end
70
71  hex_out = dec2hex(y);
72  end
```

Función *RP*

```
1   function [Y_64] = RP(X_64)
2   % This function divides a 64-bit input into eight 8-bit
        data and then
3   % permutates them in a certain way. It is used in the
        data processing
4   % part of PICCOLO.
5
6   X0 = X_64(1:8); X1 = X_64(9:16); X2 = X_64(17:24); X3 =
        X_64(25:32);
7   X4 = X_64(33:40); X5 = X_64(41:48); X6 = X_64(49:56); X7
        = X_64(57:64);
8
9   Y_64 = [X2,X7,X4,X1,X6,X3,X0,X5];
10
11  end
```

Función *RK*

```
1   function [rki] = RK(rk,i)
2   % This functions takes the part of the total round key
        that corresponds to
3   % the index i.
4   rki = rk(16*i + 1 : 16*(i+1));
5   end
```

Función *Piccolo80*

```
1   function [Ciphertext_bin] = Piccolo80(Plaintext_bin,
        Key_bin)
2
3   % Plaintext_bin = hexToBinaryVector(Plaintext,64);
4   % Key_bin = hexToBinaryVector(Key,80);
5   r = 25;
6
7   [wk,rk] = KS80(r,Key_bin);
8
9   Ciphertext_bin = Gr_encrypt(Plaintext_bin,wk,rk,r);
10
11  % Ciphertext = binaryVectorToHex(Ciphertext_bin);
12
```

```
13 end
```

Anexo II: Imágenes de la ejecución de la aplicación

```
>> AppCryptoTest
-----------------GENERADOR DE VECTORES DE TEST----------------------
Bienvenid@ a la aplicación de testeo de algoritmos criptográficos.
Esta aplicación está diseñada para generar vectores de testeo de
forma automática y pseudoaleatoria.
--------------------------------------------------------------------
De momento se encuentra implementado Piccolo-80.
--------------------------------------------------------------------
Opciones:
Pulse 1 para ---> Generar vectores de test de Piccolo-80
Pulse 2 para ---> Configurar un nuevo algoritmo en la aplicación
Pulse 3 para ---> Generar vectores de test del algoritmo que acaba
                  de configurar en la opción 2
¿Qué opción desea ejecutar? (1, 2 o 3) :  1
Ha seleccionado Piccolo-80
--------------------------------------------------------------------
Seleccione el número de vectores test que desea generar:   5
--------------------------------------------------------------------
Por favor, espere mientras se ejecuta el test...
--------------------------------------------------------------------
Gracias por esperar, su testeo ha sido realizado con éxito
Puede encontrar los datos obtenidos en tres ficheros .mat
situados en la ventana "Current Folder"
--------------------------------------------------------------------
>>
```

Figura 9: Ejecución de la opción 1 de la aplicación.

```
function [ciphertext] = Algoritmo_nuevo(plaintext,key)

% BIENVENID@, está usted aquí porque desea generar vectores de test de un
% nuevo algoritmo.

% Por favor, sustituya el nombre 'Nuevo_Algoritmo' por el nombre del script
% que contiene el algoritmo que desea testear:
%               |   |   |
%               |   |   |
%               |   |   |
%               V   V   V
ciphertext = NewALG(plaintext,key);

% Una vez haya terminado, guarde los cambios pulsando 'SAVE' y vuelva a la
% ventana de comandos (Command Window). Debe reiniciar la aplicación y
% pulsar la opción 3.
end
```

Figura 10: Introducción de un nuevo algoritmo.

```
>> AppCryptoTest
-----------------GENERADOR DE VECTORES DE TEST----------------------
Bienvenid@ a la aplicación de testeo de algoritmos criptográficos.
Esta aplicación está diseñada para generar vectores de testeo de
forma automática y pseudoaleatoria.
--------------------------------------------------------------------
De momento se encuentra implementado Piccolo-80.
--------------------------------------------------------------------
Opciones:
Pulse 1 para ---> Generar vectores de test de Piccolo-80
Pulse 2 para ---> Configurar un nuevo algoritmo en la aplicación
Pulse 3 para ---> Generar vectores de test del algoritmo que acaba
                  de configurar en la opción 2
¿Qué opción desea ejecutar? (1, 2 o 3) :  2
Acaba de configurar un nuevo algortimo, para testearlo, vuelva a
ejecutar la aplicación y en este caso pulse la opción 3
>>
```

Figura 11: Mensaje de la aplicación tras introducir un nuevo algoritmo.

```
>> AppCryptoTest
-----------------GENERADOR DE VECTORES DE TEST----------------------
Bienvenid@ a la aplicación de testeo de algoritmos criptográficos.
Esta aplicación está diseñada para generar vectores de testeo de
forma automática y pseudoaleatoria.
--------------------------------------------------------------------
De momento se encuentra implementado Piccolo-80.
--------------------------------------------------------------------
Opciones:
Pulse 1 para ---> Generar vectores de test de Piccolo-80
Pulse 2 para ---> Configurar un nuevo algoritmo en la aplicación
Pulse 3 para ---> Generar vectores de test del algoritmo que acaba
                  de configurar en la opción 2
¿Qué opción desea ejecutar? (1, 2 o 3) :  3
Ha seleccionado introducir un nuevo algortimo
--------------------------------------------------------------------
Seleccione el número de vectores test que desea generar:   4
Longitud del bloque de plaintext (en bits):   64
Longitud de la clave (en bits):   128
--------------------------------------------------------------------
Por favor, espere mientras se ejecuta el test...
--------------------------------------------------------------------
Gracias por esperar, su testeo ha sido realizado con éxito
Puede encontrar los datos obtenidos en tres ficheros .mat
situados en la ventana "Current Folder"
--------------------------------------------------------------------
>>
```

Figura 12: Ejecución de la opción 3 de la aplicación.

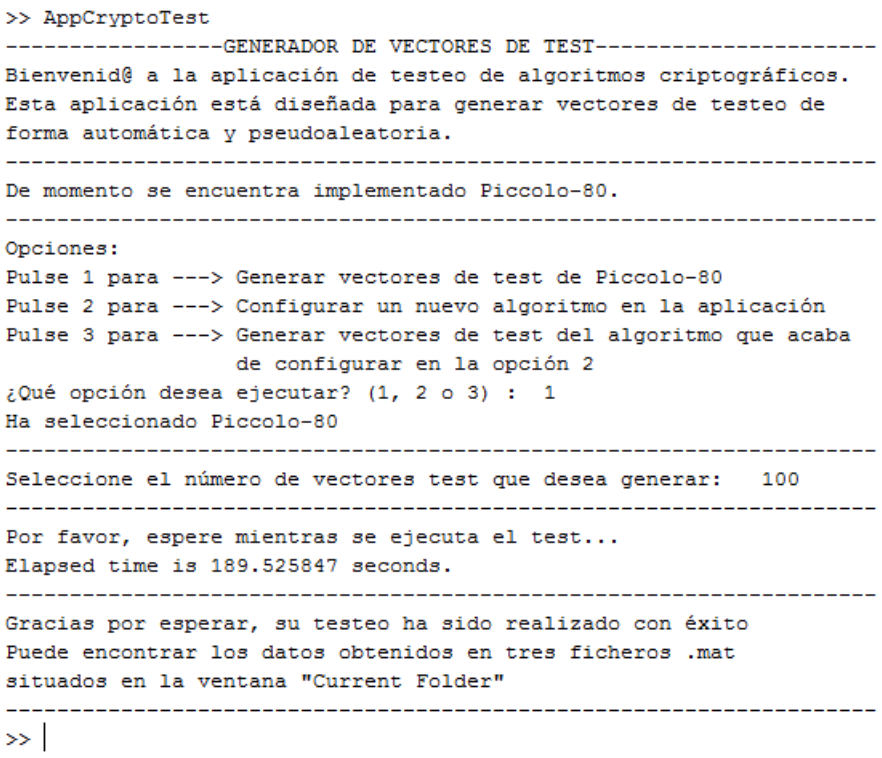

Figura 13: Medición del tiempo de generación de 100 vectores test de *Piccolo-80*.