Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica

Implementación y análisis de algoritmos criptográficos en un FPGA.

Por:

Alejandro León Torres

Ciudad Universitaria "Rodrigo Facio", Costa Rica

Diciembre de 2015

Implementación y análisis de algoritmos criptográficos en un FPGA.

Por:

Alejandro León Torres

IE-0499 Proyecto eléctrico

Aprobado por el Tribunal:

M.Sc. Diego Valverde Garro
Profesor guía

M.Sc. Carlos Duarte Martínez
Profesor lector

M.Sc. Enrique Coen Alfaro
Profesor lector

Resumen

El presente proyecto investiga sobre la teoría de criptografía y como la misma es implementada en la computación para el resguardo de datos, específicamente en hardware haciendo uso de FPGAs y del lenguaje de descripción de hardware Verilog.

Como punto de partida se elegirán dos algoritmos criptográficos comúnmente empleados, para llevar a cabo un análisis comparativo de una serie de parámetros que son relevantes para su implementación en una plataforma de FPGA. Estos parámetros consideran tres de las mayores limitantes para implementación de algoritmos en FPGA, como son el consumo de celdas, la cantidad de memoria interna utilizada y finalmente el uso de celdas especializadas de aritmética o DSP.

Índice general

Índice de figuras

Índice de tablas

1 Introducción

1.1 Justificación

El cifrado de datos para aplicaciones de seguridad es de vital importancia en la actualidad. Un ejemplo de esto es la fuga de datos de Home Depot en el año 2014, donde se perdió una computadora que contenía información personal de 10000 empleados y clientes causando un costo de millones de dólares a la empresa (Vance, 2008). Otro ejemplo fue la noticia en mayo del 2015 sobre el buscador web *UC Browser* donde quedó en evidencia que es posible robar datos personales por falta de cifrado en la transmisión de datos (Hamilton, 2015). También en febrero del 2015 se dio a conocer la fuga de decenas de millones de datos de la compañía de seguros *Anthem*. Los datos robados (números de seguro, lugar de residencia, números de teléfono, entre otros), se encontraban en una base de datos en la cual los datos se encontraban sin cifrar como indica Yadron y Beck (2015):

> *"Anthem descubrió que hackers irrumpieron en la base de datos y se hicieron de la información de decenas de millones de consumidores, siendo esta la mayor violación de seguridad informática en las compañías de servicios de salud. Debido a que los datos no se encontraban cifrados, eran fácilmente leíbles por los hackers."*

Dada la importancia del cifrado de datos, ahora nos debemos plantear ¿Porqué es importante cifrar datos utilizando hardware?.

En el caso del cifrado implementado en software, trae consigo problemáticas como lo son la necesidad de actualizaciones y el impacto en el desempeño del computador. Este último aspecto se debe a la necesidad de dedicar recursos del sistema para cifrar y descifrar la información (Apricorn, 2015).

En el caso de hardware se tienen los beneficios de que es muy confiable, rápido y conveniente. A diferencia del cifrado en software se tiene una parte de hardware dedicada al proceso de cifrado/descifrado y por tanto el desempeño de la computadora no se ve tan afectado (Apricorn, 2015). Otra de las grandes ventajas del cifrado basado en hardware es la facilidad de configuración, ya que gran parte de esto proceso es eliminado debido a que el hardware se encarga directamente de esto (SANS, 2007).

Debido a lo expuesto anteriormente se buscó trabajar en algoritmos criptográficos en hardware ya que es un área de trabajo que se puede explotar para investigar y mejorar.

Se eligió como plataforma de trabajo un FPGA debido al bajo costo que permite usarla como un plataforma de desarrollo conveniente, la popularidad que ha alcanzado en los últimos años y las mejoras que se le han realizado para que estos dispositivos logren desempeños similares a los ASICs.

1.2 Alcances y limitaciones del proyecto

Se implementarán dos algoritmos criptográficos en un FPGA haciendo uso de Verilog como lenguaje de descripción de hardware.

Posteriomente y mediante las herramientas de síntesis de Xilinx se realizará un análisis de métricas críticas en el desarrollo de aplicaciones en FPGAs como los son la cantidad de compuertas o celdas, el consumo de memoria interna del FPGA así como la cantidad de bloques aritméticos o de DSP que son usados por cada algoritmo. Uno de los factores que aunque tiene importancia no se tomará en cuenta es el consumo de potencia.

Inicialmente se va llevará a cabo un análisis individual de cada algoritmo, haciendo uso de las métricas anteriormente descritas y variando parámetros comunes de los algoritmos criptográficos como lo son el tamaño de la llave y la cantidad de rondas de cifrado (este último en algoritmos de tipo Feistel).

Como segunda parte del proyecto se realizará un análisis comparativo entre ambos algoritmos eligiendo parámetros fijos para ambos.

Los análisis individuales y comparativos anteriomente mencionados no abarcarán ningún tipo de criptoanálisis de algún algoritmo con respecto otro ni de cuál sería la escogencia de los parámetros ideal para realizar un análisis comparativo entre ambos. El enfoque más bien será que a partir del análisis individual realizado se va a efectuar una escogencia de los parámetros de ambos algoritmos para realizar su comparación implementando las métricas descritas.

Para la escogencia de estos dos algoritmos se realizará a partir de una identificación de las principales ramas del cifrado para así elegir los dos algoritmos de dos de estas ramas abarcando de esta manera un tema más amplio para el análisis y discusión.

La contribución de este trabajo se limitará a realizar una escogencia de esta manera donde también tendrá mucho peso que los algoritmos que sean implementables, de manera relativamente sencilla, en un FPGA conociendo desde un principio las limitantes estáticas del hardware.

1.3 Objetivos

Objetivo General

Implementar dos algoritmos de cifrado en un FPGA y realizar un análisis comparativo de la implementación de ambos algoritmos, empleando una serie de parámetros previamente seleccionados.

Objetivos Específicos

1. Implementar dos algoritmos criptográficos comúnmente empleados en un FPGA utilizando el lenguaje de descripción de hardware Verilog.
2. Realizar un análisis individual para cada uno de los algoritmos, variando alguno de sus parámetros (por ejemplo el tamaño de la llave) para comparar haciendo uso de métricas como la cantidad de compuertas, el consumo de memoria interna del FPGA así como la cantidad de bloques aritméticos o de DSP que se van a ir utilizando conforme se varíe el parámetro del algoritmo elegido.
3. Realizar un análisis comparativo de los dos algoritmos implementados, utilizando como métricas la cantidad de compuertas o celdas, el consumo de memoria interna del FPGA así como la cantidad de bloques aritméticos o de DSP que son usados por cada algoritmo.

1.4 Metodología

La metodología que se siguió para la realización del proyecto es la siguiente:

1. Estudios bibliográficos de:
    - Criptografía: importancia y como la misma se implementa en la computación.
    - Algoritmos criptográficos: Identificación de ramas y subramas.
    - Código e implementación de algoritmos criptográficos en diferentes lenguajes de programación.
2. Escogencia de los algoritmos criptográficos a implementar.
3. Implementación de los algoritmos criptográficos en un FPGA de Xilinx.
4. Realización del análisis individual de cada uno de los algoritmos.
5. Realización del análisis comparativo entre ambos algoritmos.
6. Realización de las conclusiones y Recomendaciones.

1.5 Desarrollo

El presente informe se estructura para el lector de la siguiente manera:

1. Capítulo I: Introducción.
2. Capítulo II: Antecedentes y Marco Teórico.
3. Capítulo III: Implementación de los algoritmos criptográficos en el FPGA.
4. Capítulo IV: Resultados de los análisis individuales y comparativos de los 2 algoritmos implementados en el FPGA.
5. Capítulo V: Conclusiones y recomendaciones.

2 Marco Teórico

Este capítulo va a poner en contexto al lector con respecto a los conceptos básicos de criptografía, va a definir y explicar los algoritmos que van a ser implementados en el FPGA. Además va a realizar una breve explicación sobre características de un FPGA para explicar el porqué de la escogencia de los parámetros que van a ser analizadas.

2.1 Conceptos Básicos

Según la RAE (2015a) la criptografía se define como

> *Arte de escribir con clave secreta o de un modo enigmático.*

El mensaje que se desea transmitir es llamado *texto plano* o simplemente *mensaje*. Este mensaje pasa por un proceso donde se disfraza el texto plano en un *texto cifrado*, el proceso se denomina *cifrado*[1]. El proceso inverso donde se toma un texto cifrado en un texto plano se denomina *descifrado* (Schneier, 1996).

El texto plano o mensaje se denota por la letra M, el texto cifrado se denota por la letra C, la función o algoritmo que cifra se denota por E y la que descrifa se denota por D. Un algoritmo criptográfico corresponde a la función matemática para cifrar y descrifrar.
Se muestra en las Ecuaciones (2.1) y (2.2) las relaciones entre estas notaciones. Note como al aplicarle la función de cifrado al texto plano se obtiene el texto cifrado y como al aplicarle la función de descifrado al texto cifrado se obtiene el texto plano. Finalmente se debe cumplir la identidad que describe la Ecuación (2.3) (Schneier, 1996).

$$E(M) = C \tag{2.1}$$

$$D(C) = M \tag{2.2}$$

$$D(E(M)) = M \tag{2.3}$$

La importancia de la criptografía transciende más allá de brindar la confidencialidad en la comunicación, la criptografía también cumple con la siguientes tareas:

[1]Según el ISO 7498-2 los términos correctos para encriptar y desencriptar son cifrar y descifrar respectivamente.

- Autenticación: El receptor del mensaje debe de poder conocer y asegurar el emisor del mensaje, esto para que un tercero no pueda adjudicarse la identidad del emisor.

- Integridad: El receptor tiene que poder asegurarse que el mensaje no fue cambiado en el transito del mismo. Esto para que un tercero no pueda cambiar el contenido enviado por el emisor sin que el receptor lo sepa.

- *Non-repudiation*: El emisor del mensaje no puede negar que el mensaje fue enviado por él.

2.2 Sistema criptográfico (*criptosistema*)

Cuando la seguridad del algoritmo se basa en como procede el algoritmo, se denomina *algoritmo restringido*. Este tipo de algoritmos son poco utilizados en la actualidad debido al gran problema que presentan. Tomemos de ejemplo que un grupo de usuarios decide utilizar un algoritmo criptográfico restringido para sus comunicaciones, se tendrá una comunicación segura hasta que alguno de los miembros decida salirse del grupo, ya que el usuario al no pertenecer más al grupo, no le importa mantener en secreto el algoritmo y puede distribuirlo para que terceros intercepten las comunicaciones. Así cada vez que un miembro deja el grupo, se debe proceder a cambiarse a todo un nuevo algoritmo lo cual puede tornarse una labor complicada. También ocurre el problema que si se logra obtener un algoritmo equivalente para cifrar los datos, se debe migrar a un nuevo algoritmo.

En cambio, la criptografía moderna (Denning, 1982) agrega el concepto de *llave* donde se tiene un algoritmo el cual toma como parámetros de entrada una llave y el texto plano o cifrado, y cifra o descifra el mismo de forma correcta, únicamente si se tiene la llave correcta. El esquema descrito anteriormente se muestra en la Figura 2.1. Retomando el ejemplo anterior, el grupo solamente necesitaría cambiar de llave cuando un miembro se va, facilitando el uso del cifrado y manteniendo las comunicaciones secretas.

Estos conceptos viene a definir lo que actualmente se conoce como sistema criptográfico o *criptosistema*. Según Denning (1982), un criptosistema cuenta con 5 componentes:

- Un espacio de textos planos o mensajes (M)

- Un espacio de textos cifrados (C)

- Un espacio de llaves (k)

- Una familia de *transformaciones de cifrado*: $E_K : M \to C$ donde $K \epsilon k$

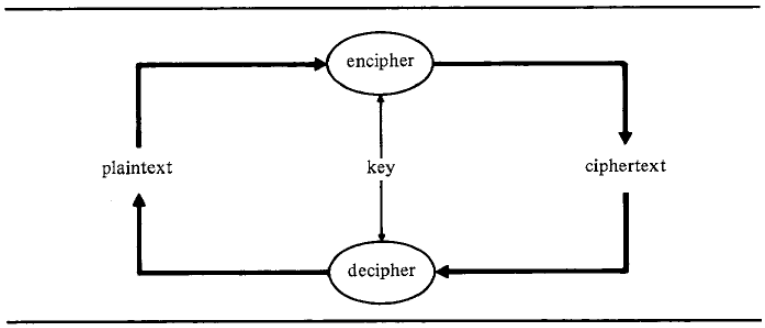

Figura 2.1: Descripción gráfica de un sistema criptográfico (Denning, 1982).

- Una familia de *transformaciones de descrifrado*: $C_K : C \rightarrow M$ donde $K \epsilon k$

Se entiende como *espacio* el conjunto de posibles valores para la variable dada, sea esta M, C o K. Y una familia de transformaciones corresponde a todos los posibles mapeos que se pueden realizar de un espacio a otro (de M a C o viceversa) con todos los valores contenidos en el espacio k.

En la actualidad se trabaja la criptografía sobre computadoras, es decir, cifrando y descrifrando bits, los cuales pueden tener diferentes significados, ya sea una imagen, un texto, un programa, etc. Esto significa que en la actualidad no se trabaja sobre caracteres del alfabeto o símbolos, sino más bien sobre 1's y 0's. Esto toma relevancia ya que al tener solo dos símbolos para cifrar, los algoritmos se vuelven más complejos por la falta de alternativas para sustituir un símbolo por otro.

Antiguamente la criptografía se basaba en caracteres que eran sustituidos o traspuestos por otros caracteres. Esto corresponde a cifrados de sustitución y de transposición, los cuales continúan siendo la base de la criptografía pero basado en los dos símbolos del sistema binario.

Como se explicó anteriormente el cifrado de sustitución se basa en tomar un caracter del texto plano y sustituirlo por otro caracter. Para descrifrar el texto cifrado simplemente se sustituyen de vuelta los caracteres y listo.

Según Schneier (1996), en la criptografía clásica existen 4 tipos de cifrado por sustitución:

- Cifrado de sustitución simple: Una sustitución de uno a uno entre cada caracter del texto plano y el texto cifrado. Ejemplos de este tipo de sustitución son el famoso cifrado de César y el ROT13 utilizado en UNIX.

- Cifrado de sustitución homofónico: Una sustición de uno a muchos. Un caracter del texto plano, por ejemplo A, puede ser sustituido por varios

Letter	Homophones
A	17 19 34 41 56 60 67 83
I	08 22 53 65 88 90
L	03 44 76
N	02 09 15 27 32 40 59
O	01 11 23 28 42 54 70 80
P	33 91
T	05 10 20 29 45 58 64 78 99

One possible encipherment of the message is:

M =	P	L	A	I	N	P	I	L	O	T
C =	91	44	56	65	59	33	08	76	28	78

Figura 2.2: Ejemplo de cifrado homofónico (Denning, 1982).

caracteres en el texto crifado, por ejemplo “5”, “13”, “43”. Observe la Figura 2.2 donde se presenta una serie de posibles asignaciones de números a las letras del mensaje PLAIN PILOT y un posible texto cifrado haciendo uso de este tipo de cifrado.

- Cifrado de sustitución de poligrama: Una sustitución por bloques en donde se toma un bloque de caracteres del texto plano y se sustituye por su bloque equivalente en el texto cifrado. Por ejemplo si en el texto plano se tiene “ABC” se sustituye por “SLL” en el texto cifrado.

La otra variedad de algoritmos son los de transposición, en este tipo de algoritmos criptográficos el texto plano se convierte en texto cifrado cuando el orden de los caracteres es cambiado bajo alguna norma. Un ejemplo moderno de este tipo de algoritmos es el *rail-fence* donde el texto plano se reacomoda con la forma de una cerca como se observa en la Figura 2.3. En este caso la llave del algoritmo sería la profundidad de la cerca, para efectos de este ejemplo es de 3.

Actualmente en los algoritmos que se implementan en computadoras se combina tanto la transposición como la sustitución. Por ejemplo se tiene el algoritmo RC5 (el cual se desarrollará más adelante en este proyecto), en donde se utiliza corrimientos o rotaciones a bits (transposición) y sumas o XOR's (sustituciones) para cifrar el texto plano. Los algoritmos que se implementan en la criptografía moderna se dividen en dos categorías principales: simétricos y de llave pública (también llamados asimétricos (Denning, 1982)).

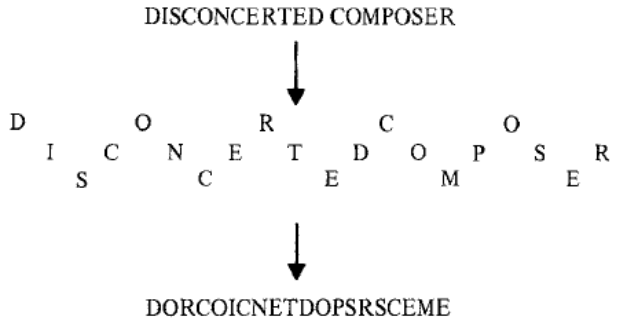

Figura 2.3: Ejemplo de cifrado de transposición (Denning, 1982).

2.3 Algoritmos criptográficos

Algoritmos simétricos

Schneier (1996) da una muy buena analogía para explicar el concepto de un algoritmo simétrico. Piense en el algoritmo como una caja fuerte. La combinación de la caja fuerte vendría a ser la *llave* del algoritmo. Note como en una caja fuerte cualquier persona con la combinación puede llegar, abrir la caja y poner o sacar documentos de la misma. En el caso de no conocer la combinación, se debe proceder a forzar la caja o probando todas las combinaciones posibles hasta hallar la correcta. Es decir en un algoritmo simétrico se cuenta con una llave única que funciona tanto para cifrar como para descrifrar los mensajes como se muestra en la Figura 2.4. La notación para el cifrado y descrifado en estos algoritmos se muestra en las Ecuaciones (2.4) y (2.5).

$$E_K(M) = C \tag{2.4}$$

$$D_K(C) = M \tag{2.5}$$

Los algoritmos simétricos se dividen en 2 categorías (Schneier, 1996):

- Cifrado de bloque: Se cifra en bloques de bits ya sea *bytes, words*, etc. Es decir cuando se va a proceder a cifrar un texto plano, se segmenta el texto en grupos de bits y estos son cifrados en conjunto. Se puede tomar como ejemplo el algoritmo *Data Encryption Standart* (DES) el cual cifra sobre bloques de 64 bits y es uno de los algoritmos simétricos más importantes. Otros ejemplos son:
  - Lucifer.
  - LOKI.

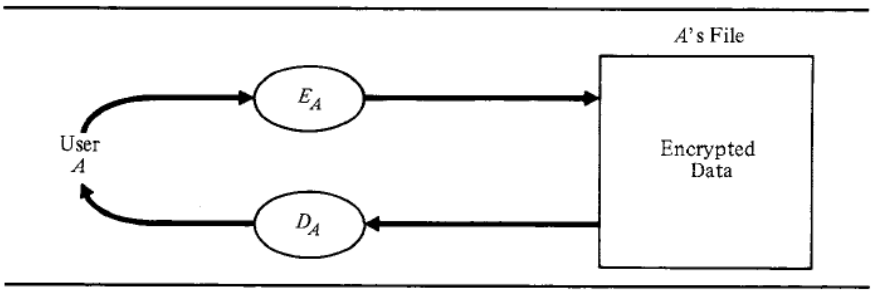

Figura 2.4: Descripción gráfica de una algoritmo de llave simétrica (Denning, 1982).

  - 3-way.
  - RC5.
  - GOST.
  - IDEA.

- Cifrado de *Stream*: Es cuando se trabaja sobre un bit únicamente. Estos no son muy usuales en la actualidad ya que al trabajar en lenguaje binario solo se cuenta con 2 símbolos y si se cifra únicamente un bit no existen muchas posibilidades para sustituir o transposicionar. Ejemplos de estos algoritmos pueden ser:

  - RC4.
  - SEAL.
  - WAKE.

Según Schneier (1996), los criptosistemas simétricos en la red afrontan los siguientes problemas

- Distribución de la llave: La llave se debe mantener en secreto. Esto en la actualidad es una tarea demasiado díficil de lograr porque la llave debe ser conocida para el cifrado y descifrado (emisores y receptores) entonces para establecer una comunicación segura el primer paso debe ser entregar la llave de forma segura, lo cual en una red de computadoras se puede tornar una tarea prácticamente imposible de realizar. La única solución sería entregar las llaves mediantes un servicio de *courier* o similares e igualmente se corren riesgos.

- Compromiso de seguridad: Si la llave es conocida por un tercero, si este intercepta el tráfico de información, todas las comunicaciones serán descifradas fácilmente.

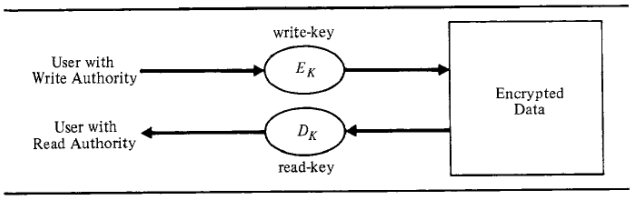

Figura 2.5: Descripción gráfica de una algoritmo de llave pública (Denning, 1982).

- Comunicaciones aisladas: En el caso de que cada usuario de una red se desee comunicar secretamente por separado con todos los otros usuarios haciendo uso del mismo criptosistema, se debe utilizar una llave diferente para cada comunicación. Así para una red de N usuarios se requieren $N(N-1)/2$ llaves. A primera vista esto no parece tan importante por ejemplo para 10 usuarios, se necesitan 45 llaves lo cual está bien, pero para 100 usuarios se necesitarían 4950 llaves que se tienen que distribuir de forma secreta.

Algoritmos de llave pública

Nuevamente Schneier (1996), nos ofrece una excelente analogía para explicar, en este caso, los algoritmos de llave pública. Tenemos un *mailbox*, donde cualquier persona puede poner un mensaje adentro pero **únicamente** el dueño puede abrirlo para sacar y leer los mensajes.

En los algoritmos de llave pública los emisores hacen uso de una llave para cifrar los mensajes que desean enviar pero estos mensajes pueden ser descrifrados únicamente si se tiene la llave para descifrar mensajes (que es diferente de la llave para cifrar), la cual el receptor la tendrá bien resguardada. En la Figura 2.5 se muestra el diagrama básico de un algoritmo de llave pública. Ejemplos de estos algoritmos pueden ser:

- RSA
- Pohlig-Hellman
- Rabin
- ElGamal

Estos algoritmos sientan su base matemática en funciones llamadas *one-way* en donde al aplicarle una función a una variable, la variable no puede

retornar a su valor original de ninguna manera, es decir la función no tiene una inversa.

Según Schneier (1996), existen funciones *one-way* con una "puerta trasera" donde se puede retornar a la variable original conociendo ciertos parámetros, este tipo de funciones son las que se implementan en algoritmos de llave pública. Los algoritmos dominantes de esta rama se basan en la dificultad de factorizar números grandes que son el resultado de multiplicar dos números primos grandes como también se basan en el *Discrete Logarithm Problem*.

En estos algoritmos no es posible a partir de la llave para cifrar obtener la llave para descrifrar. Esto permite que la llave para cifrar se pueda hacer pública, por lo cual recibe el nombre de llave pública y la llave para descrifrar se denomina llave privada. La notación para estos algoritmos corresponde a la de las Ecuaciones (2.6) y (2.7)

$$E_{K_x}(M) = C \tag{2.6}$$

$$D_{K_y}(C) = M \tag{2.7}$$

El objetivo de utilizar cifrado de llave pública se basa en:

- Receptor y emisor acuerdan un sistema criptográfico.
- El receptor entrega al emisor su llave pública.
- El emisor cifra el texto plano haciendo uso de la llave pública entregada y el sistema acordado.
- El emisor envía el texto cifrado.
- El receptor descrifa el texto cifrado haciendo uso de la llave privada.

De esta manera no hay forma en que fisgones logren descrifrar el mensaje aunque obtengan la llave pública. Así el receptor se asegura que las comunicaciones van a ser mucho más seguras, ya que el emisor deja de tener la llave para descifrar y no puede brindarsela a nadie o que le sea robada a este. Para la implementación de un criptosistema que hace uso de algoritmos de llave pública, lo que se hace es que en la red se tiene una base de datos donde se registra el usuario y su respectiva llave pública. Así cuando un usuario desea comunicarse con otro, va a la base de datos, busca al usuario y su llave pública, cifra el mensaje con la misma y se la envía. Esto solventa los problemas que presentaba anteriormente los algoritmos simétricos, ya que no es necesario transmitir de forma secreta llaves para poder realizar comunicaciones y para que diferentes usuarios se comuniquen de forma secreta entre si no es necesario el uso de diferentes llaves por cada enlace de comunicación.

Algunos usos de estos algoritmos en la actualidad son:

- Llave maestra para el sistema de pago digital de un banco.
- La clave que utiliza un gobierno para certificar sus visas o pasaportes.
- La firma digital de un notario público.

Criptosistemas híbridos

Los beneficios que conlleva utilizar algoritmos de llave pública son grandes pero se pagan a un precio muy alto: tiempo de procesamiento. Según Schneier (1996), el tiempo de procesamiento del RSA con respecto al del DES es de alrededor de 1000 mil veces más lento.

En un mundo donde la velocidad es una clave fundamental en las comunicaciones se propuso la siguiente solución. Ya que los algoritmos simétricos tienen la debilidad de comunicar la llave antes de comenzar la comunicación pero son mucho más rápidos, y que los algoritmos de llave pública ostentan una mejor sistema para comunicarse en la red secretamente pero son muy lentos, se decidió utilizar ambos. La llave del algoritmo simétrico es cifrado con un algoritmo de llave pública para transmistirse en la red sin comprometerla y posterior a esto se realizan las comunicaciones con el algoritmo simétrico para obtener una mejor velocidad de comunicación. Se puede ver el protocolo de la siguiente manera (Schneier, 1996):

- El receptor envía su llave pública al emisor.
- El emisor genera una llave de sesión[2], la cifra y se la envía al receptor usando la llave pública que le fue dada.
- El receptor descrifra la llave de sesión usando su llave privada.
- Ahora ambos pueden comunicarse de forma secreta con la llave de sesión con un criptosistema simétrico.

2.4 Seguridad en algoritmos criptográficos

Como parte de la investigación previa para este proyecto no podemos dejar de lado la otra cara de la moneda, el criptoanálisis. La ciencia que abarca la criptografía y el criptoanálisis es conocida como criptología (Denning, 1982). Claramente el objetivo de la criptografía es mantener un mensaje en secreto de

[2]Una llave de sesión se utiliza en comunicaciones donde la idea es cifrar cada comunicación de manera individual con una llave distinta. Son útiles cuando la llave se crea al inicio de la comunicación y se destruye al final de la misma (Schneier, 1996).

terceros, pues el criptoanálisis según RAE (2015b), es el "arte de descifrar criptogramas", así es como los terceros buscan inmiscuirse en las comunicaciones secretas sin tener un acceso a la llave.

Según Schneier (1996), el criptógrafo al diseñar su algoritmo debe asumir que el criptoanalista puede tener un acceso completo a las comunicaciones entre emisor y receptor y al algoritmo que implementa el criptosistema, así toda la seguridad del criptosistema debe residir en la llave.

La acción en la que un criptoanalista atenta contra un criptosistema es denominada *ataque*. Según Schneier (1996); Denning (1982), los principales tipos de ataques son:

- *Ciphertext-only*: El criptoanalista debe obtener la llave a partir de varios textos cifrados. Debe conocer el tema tratado de las comunicaciones que está interviniendo (lo cual es obvio, ya que sino para qué está interviniendo las comunicaciones) entonces puede saber de forma previa que ciertas cosas que puede contener el texto plano.

- *Known-plaintext*: El criptoanalista tiene acceso a ciertos textos planos y sus respectivos textos cifrados. A partir de estos debe deducir la llave o debe generar un algoritmo de descifrado equivalente al diseñado por el criptográfo.

- *Chosen-plaintext*: El criptoanalista puede obtener el texto cifrado de un texto plano que él seleccione, esto es mucho más poderoso porque puede cifrar mensajes que den más información acerca de la llave. Un sistema de bases de datos es vulnerable a este tipo de ataques debido a que los usuarios pueden agregar elementos a la base de datos y ver el resultado del texto cifrado en la misma. Nuevamente su tarea es obtener la llave o generar un algoritmo de descifrado equivalente.

- *Chosen-ciphertext*: Este tipo de ataque se da en algoritmos de llave pública, en donde el criptoanalista tiene acceso al texto cifrado que va a ser descifrado y al texto plano. Es decir tiene una "caja negra" donde la entrada es el texto cifrado y la salida el texto plano. El objetivo es obtener la llave a partir de estos recursos.

- *Chosen-key*: El criptoanalista tiene conocimiento sobre la relación entre las diferentes llaves.

- *Rubber-hose*: El criptoanalista amenaza, extorsiona, chantajea u obtiene de alguna forma la llave sin ningún tiempo.

Otro ejemplo es que los algoritmos de llave pública son vulnerables a ataques de tipo *chosen-plaintext*. Tome por ejemplo M como un texto plano del

espacio M de posibles mensajes. La tarea de un criptoanalista es tomar todos MϵM y cifrarlos para obtener todos los posibles CϵC. De esta manera solo debe interceptar las comunicaciones que desea y comparar sus valores de texto cifrado con los que intercepta para descifrar las comunicaciones. Note que el criptoanalista no pudo obtener la llave pero si logró descrifrar las comunicaciones.

Tamaño de la llave

Como se mencionó anteriormente la seguridad de un buen algoritmo depende de su llave. Asumiendo que se tiene una seguridad del algoritmo perfecta, la única forma de quebrar el criptosistema sería mediante un ataque de fuerza bruta, el cual es un tipo de ataque *known-plaintext*, para obtener la llave.

Analizando el tamaño de la llave, si se tuviera una de 2^8 bits, probando las 256 posibilidades se obtendría la llave, lo cual equivale a unos cuantos segundos de procesamiento en una computadora. Para 2^{64} bits tenemos $1{,}8446744^{19}$ posibles llaves, lo cual tomaría con los recursos computacionales de una supercomputadora alrededor de 585,000 años. Para una llave de 2048, con un billón de intentos por segundo en computadoras en paralelo se necesitarían 10597 años para encontrar la llave (Schneier, 1996).

Entonces ¿Porqué no hacer un algoritmo con una llave muy grande?. La respuesta es porque conforme aumenta el tamaño de la llave, se paga en tiempo de procesamiento. Así se debe obtener un balance donde la llave sea lo suficientemente grande para que sea el algoritmo sea seguro, pero lo suficientemente pequeña para que el tiempo de procesamiento sea bueno (Schneier, 1996).

Comparar el nivel de seguridad ante un ataque de fuerza bruta de un algoritmo simétrico y uno de llave pública con llaves del mismo tamaño no es posible, pero la Figura 2.6 muestra una tabla con equivalentes realizados empíricamente por Schneier (1996) sobre tamaños de llaves que dan un nivel de seguridad equivalente para los dos tipos de algoritmos, esto nos indica que no es recomendable comparar algoritmos de distintas ramas entre sí.

Manejo de las llaves

Se puede tener un algoritmo extremadamente robusto, veloz y con un tamaño de llave perfecto, pero si un atacante puede obtener la llave por algún medio el algoritmo es completamente inútil. De ahí la gran importancia de como manejar las llaves.

Al ser esto tan importante existen ciertas maneras de generar llaves:

- Ingresada por el usuario: El usuario elige una llave y esa es la que se va a utilizar. Este método es sumamente inseguro debido a que gran

Symmetric Key Length	Public-key Key Length
56 bits	384 bits
64 bits	512 bits
80 bits	768 bits
112 bits	1792 bits
128 bits	2304 bits

Figura 2.6: Comparación del tamaño de llaves en algoritmos simétricos y asimétricos (Schneier, 1996).

cantidad de contraseñas se repiten, o son datos personales del usuario como su número de celular o similares. Por tanto se pueden realizar *ataques de diccionario* donde las primeras llaves que se prueban son las mencionadas anteriormente.

- *Random keys*: Es un muy buen método para generar llaves, consiste en utilizar un programa que genere la llave, es ventajoso ya que es robusto ante ataques de diccionario pero presenta el problema que la llave es difícil de recordar y posiblemente se olvide.

- *Pass phrases*: Es una combinación de ambos, el usuario escribe una contraseña fácil de recordar y después hace uso de una función *one-way* el sistema convierte esta llave en una llave random de tamaño arbitrario.

2.5 *Field Programmable Gate Array* (FPGA)

Un FPGA consiste en una matriz de bloques lógicos configurables, conectados mediantes interconexiones programables. Posterior al proceso de manufactura el FPGA puede ser reprogramado para darle una nueva funcionalidad, esto

Tabla 2.1: Ventajas y beneficios de diseñar con FPGAs (Xilinx, 2015c).

Ventaja	Beneficio
Tiempo de producción más corto	Pasos de manufactura como layout, máscara u otros no son necesarios
No upfront non-recurring expenses (NRE)	Costo que ocurre en el diseño de ASICs.
Ciclo de diseño más simple	El software se encarga en gran parte del *placing, routing* y *timing.*
Ciclo del proyecto más predecible	Debido a que se eliminan los posibles re-spins, capacidades de la oblea, etc.
Field reprogramability	Se puede cargar un nuevo RTL y listo.

Tabla 2.2: Ventajas y beneficios de diseñar con ASICs (Xilinx, 2015c).

Ventaja	Beneficio
100 % personalizados	Ya que se manufactura apegado a la especificación.
Costos unitarios más bajos	Para diseños de gran volumen.
Más pequeños	Debido a que el diseño se acoge completamente a la especificación.

es lo que diferencia a un FPGA de un *Application Specific Integrated Circuits* (ASIC) (Xilinx, 2015b).

FPGA vs ASIC

En años anteriores los FPGAs presentaban gran desventaja en cuanto a velocidad, consumo y área respecto a un ASIC y por eso aunque eran reprogramables, desarrollar aplicaciones en un FPGA solo se daba para realizar prototipos y no para un producto final. En la actualidad los FPGAs se encuentran en auge debido a que su proceso de manufactura ha mejorado mucho en los aspectos anteriormente descritos, por ejemplo un FPGA ya puede alcanzar los 500MHz, han aumentado la densidad de lógica y albergan otras funcionalidades como por ejemplo módulos de *Digital Signal Proccessing* (DSP), de tiempos y hasta de *high-speed serial* (Xilinx, 2015b).

Estos grandes avances en conjunto con un flujo de diseño mucho más fácil, barato y rápido, indican que el desarrollo en FPGAs va a convertirse en la nueva forma de diseñar hardware. La Tabla 2.1 muestra beneficios y ventajas del diseño con FPGAs y la Figura 2.7 muestra la diferencia en el ciclo de diseño de hardware con FPGAs y ASICs. Aún con todas estas ventajas los FPGAs todavía presentan ciertas desventajas con respecto a los ASICs, como se muestra en la Tabla 2.2.

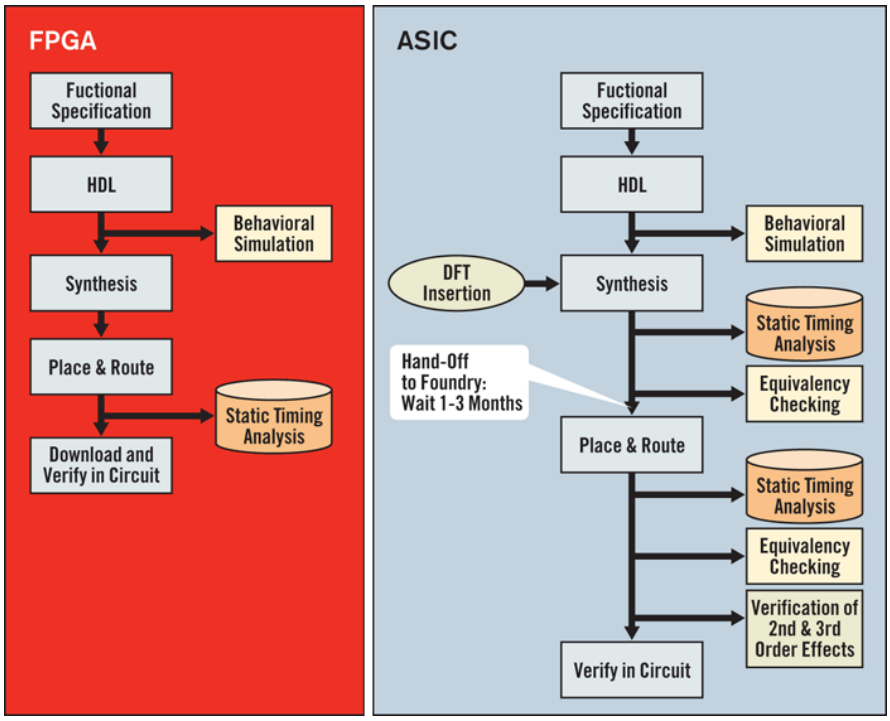

Figura 2.7: Ciclo de diseño de hardware para FPGAs y ASICs (Xilinx, 2015c).

Componentes básicos de un FPGA

Todos los FPGAs, como se muestra en la Figura 2.8, tiene como mínimo 3 componentes (Xilinx, 2011)

- *Configurable Logic Block*
- *Programmable Switching Matrix*
- *Input/Output Block*

A parte de estos componentes un FGPA también puede tener un módulos de memoria RAM, ROM y hasta módulos para DSP.

A continuación se describen de forma general los componentes que conforman los bloques descritos anteriormente (Xilinx, 2011, 2015a)

- *Configurable Logic Block* (CLB): Corresponde a la unidad principal de diseño para lógica combinatoria y flip-flops. Se encuentra compuesto por slices y la cantidad que lo componen varía según el modelo. Para determinar la densidad de slices por CLB se debe sintetizar el diseño previamente para decidir cual es el modelo que se ajusta al mejor desempeño

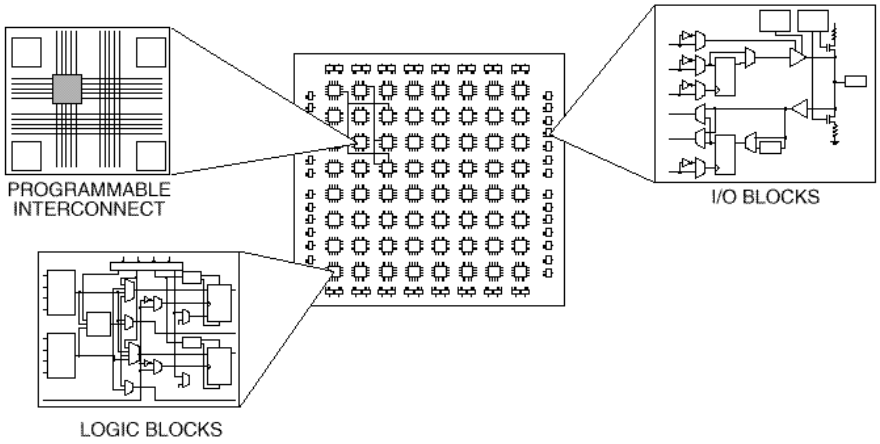

Figura 2.8: Estructura básica de un FPGA. (Egyetem, 2015)

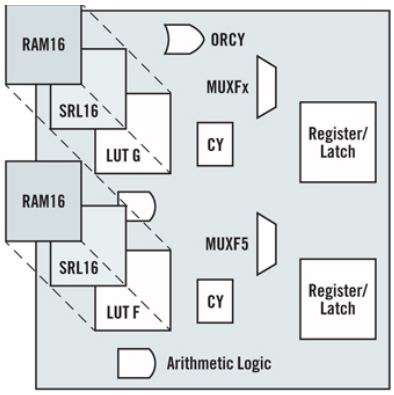

Figura 2.9: Estructura básica de un CLB (Xilinx, 2015d).

del diseño. Estos bloques son conectados a un PSM para comunicarse con otras partes del FPGA y también se conectan a un IOB para comunicarse con el mundo exterior. La Figura 2.9 muestra una representación general de lo que puede contener un CLB.

- *Slice*: Es la unidad básica de un FPGA, puede estar compuesto por:
  - *Look Up Table* (LUT): También son llamados Function Generators, depende del número de entradas y no de la complejidad de la función que se está implementando. Así el retardo es constante para todo el LUT. Son usados para implementar lógica booleano y también como memoria o SRL[3]

[3]Shift Register Lookup Table (SRL) es un registro desplazable de tamaño variable. Sus

  - *Wide Multiplexers*: Se utilizan para conectar LUTs entre si y también para no utilizar LUTs en operaciones simples, ahorrando LUTs y mejorando la velocidad de procesamiento.
  - *Carry Chain*: Cuando los CLBs deben de realizar operaciones como sumas o restas utilizan esta característica para mejorar la velocidad en la que se realiza la operación
  - *Registers*: Pueden ser flip-flops o latches.

  Ciertos slices pueden se utilizados como una memoria RAM distribuida, en donde por ejemplo el LUT se utiliza como una memoria, ya sea en configuración *single, dual, simple-dual o quad port.*

- *Programmable Switch Matrix* (PSM): Corresponde a un módulo en donde se realiza la interconexión de los distintos CLBs y otros componentes del FPGA según requiera el diseño. Esta interconexión puede realizarse de forma vertical, horizontal y diagonal.

- Input/Output Blocks (IOB): Estos bloques se encargan de comunicar al FPGA con el mundo exterior, manejan gran cantidad de estándares de comunicación como por ejemplo: LVCMOS, PCI, I2C y SSTL.

Como se puede observar los FPGAs son de gran importancia y por eso el objetivo de implementar los algoritmos criptográficos en un FPGA adquiere relevancia. Así también es muy importante conocer que tantos recursos (slices, memoria y DSP) se requieren para implementar estos diseños en un FPGA ya que al aumentar la cantidad de recursos aumentan los costos de producción, volviendo este proyecto relevante por su proyección innovadora sin dejar de lado el aspecto económico tan importante en la industria.

2.6 Escogencia de los algoritmos criptográficos a implementar

En un principio el objetivo de este proyecto era realizar una comparación con las métricas descritas en la Introducción sobre un algoritmo simétrico y un algoritmo de llave pública, posterior a la realización del marco teórico se llegó a la conclusión de que realizar este análisis no iba a ser para nada justo debido a tres razones:

- El tiempo de procesamiento: Como se menciona anteriormente los algoritmos de llave pública consumen mucho más tiempo realizando el cifrado y descifrado que un algoritmo de llave simétrica.

usos pueden ser como retardo programable o como una memoria FIFO.

- La formas en las que estos algoritmos son diseñados producen falencias en seguridad distintas, como se indica en la sección 2.4 los tamaños de llaves varían mucho para lograr una seguridad similar ante un ataque de fuerza bruta, esto haría poco justo realizar un análisis comparativo con el mismo tamaño de llave.

- Los algoritmos realizan el proceso de cifrado de maneras muy diferentes: En el caso de los algoritmos simétricos son basados en transposición y sustitución que trabajan en su mayoría con XOR's, sumas y corrimientos. En cambio los algoritmos de llave pública trabajan basados en la multiplicación de números primos y *Discrete Logarithm Problem* donde ocurren muchas multiplicaciones y sumas. Esto produce un consumo de recursos muy diferente entre ambos tipos.

Bajo el criterio que ambos algoritmos tuvieran características similares para poder ser analizados comparativamente y que fueran fáciles de implementar a nivel de hardware se optó por comparar dos algoritmos simétricos, específicamente el RC5 y el TEA los cuales están basados en redes de Feistel[4] y por tanto comparten características de diseño para que la comparación sea un poco más justa.

2.7 Descripción de los algoritmos criptográficos a implementar

RC5

El algoritmo criptográfico de llave simétrica RC5 fue creado en 1997 por Ronald Rivest para la *RSA Data Security*. Según Rivest (1997) los objetivos de este algoritmo se basaban en que fuera adaptable tanto para hardware como para software, que tuviera una llave de tamaño variable, que el tamaño de la palabra a cifrar fuera de tamaño variable para que así se ajustara a procesadores de tamaños de palabra diferentes y por último que el número de rondas de cifrado fuera variable (con la finalidad de permitir decidir que tanta seguridad y que tan rápido se cifran los datos).

[4] Las redes Feistel toman un bloque de texto plano de tamaño N (par) y lo dividen en dos mitades denominadas L y R. El cifrado en este tipo de redes es iterativo donde el resultado de la iteración i depende del resultado de la iteración $i-1$. Ejemplos de algoritmos de este tipo pueden ser DES, Lucifer, FEAL, Khufu, Khafre, LOKI, GOST, CAST, Blowfish, entre otros.

Terminología del algoritmo

Por simplicidad como se tienen tantos parámetros variables para el algoritmo entonces a la hora de llamarlo se utiliza la siguiente notación: RC5-w/r/b. Estas y otras variables se definen a continuación:

- w: es el tamaño de la palabra, el valor nominal es de 32 bits y los permitidos son 16 bits, 32 bits y 64 bits. Y el algoritmo cifra en bloques de tamaño 2w. Además cada palabra contiene $u = w/8$ bytes.
- r: corresponde al número de rondas, los cuales pueden variar entre 0 y 255. Esto debido a que el algoritmo se basa en una red Feistel.
- S: tabla con la expansión de la llave. Tiene un tamaño de $t = 2(r + 1)$ palabras.
- llave: Se tiene dos parámetros para definirla:
  - b: cantidad de bytes que tiene la llave.
  - K[i]: K-ésimo byte de la llave va desde K[0] hasta K[b-1].
- A y B: Corresponden a las partes izquierda y derecha del texto plano respectivamente.
- L: Arreglo de $c = max(b, 1)/u$ palabras que funciona para convertir la llave de bytes a palabras. Inicialmente se encuentra lleno de ceros.
- P_w y Q_w: Constantes que se definen como:

$$P_w = odd(2^w(e - 2));\ e = \text{número de euler} \tag{2.8}$$

$$Q_w = odd(2^w(\phi - 2));\ \phi = \text{golden ratio} \tag{2.9}$$

Según Rivest (1997) los beneficios de utilizar el RC5 como algoritmo de cifrado se tiene una serie de parámetros variables los cuales conforme pasa el tiempo se pueden adaptar a la necesidades de cifrado. Un ejemplo de lo necesario que es esto es el problema con el algoritmo DES , para cuando fue creado su tamaño de llave era ideal pero ahora es muy corta y no hay forma fácil de que el algoritmo acepte una más fácil.

Otros usos del algoritmo es por ejemplo (Rivest, 1997) RC5-32/8/0, es decir si el uso de una llave, para generar una secuencia de números pseudo-aleatorios.

También es importante destacar que:

- + indica suma complemento a dos. Así como - indica resta en complemento a dos.

- $\oplus$ indica un XOR bit a bit.
- <<< indica rotación hacia la izquierda, igualmente >>> indica rotación a la derecha. Entonces X <<< Y significa que la palabra X se rota Y bits a la izquierda.

Pasos del algoritmo

A continuación se muestra el pseudo-código extraído de Rivest (1997) para cifrar y descifrar datos con este algoritmo.

- Expansión de llave: Corresponde la primer paso antes de poder cifrar o descifrar texto plano. Recibe como entrada una llave de tamaño arbitrario y realiza lo siguiente:

  Listado 2.1: Pseudo-código del paso de expansión de llave en el algoritmo RC5 (Rivest, 1997).

```
for i = b - 1 downto 0 do
  L [i/u] = (L[i/u] <<< 8) + K[i]

S[0] = Pw;

for i = 1 to t-1 do
  S[i] = S[i-1] + Qw;

i = j =0
A = B =0

do 3*max(t,c) times:
  A = S[i] = (S[i] + A + B) <<< 3
  B = L[j] = (L[j] + A + B) <<< ( A + B )
  i = (i+1) mod(t)
  j = (j+1) mod(c)
```

  Esto lo realiza para transformar la llave de bytes a palabras que se guardan en el arreglo S para que cada palabra de este arreglo se utilice para cifrar o descifrar el texto plano.

- Cifrado: se realiza sobre los bloques de texto plano A y B simultáneamente, se realizan operaciones XOR y rotaciones entre el texto plano y palabras del arreglo S:

  Listado 2.2: Pseudo-código del paso de cifrado en el algoritmo RC5 (Rivest, 1997).

```
A = A + S[0];
B = B + S[1];
```

```
3
4 for i = 1 to r do
5   A = ((A ⊕ B) <<< B) + S[2i];
6   B = ((B ⊕ A) <<< A) + S[2i + 1];
```

- Descifrado: se realiza sobre los bloques cifrados A y B simultáneamente y se aplica las operaciones inversas a las del cifrado:

  Listado 2.3: Pseudo-código del paso de descifrado en el algoritmo RC5 (Rivest, 1997).

```
1 for i = r downto 1 do
2   B = ((B - S[2i + 1]) >>> A) ⊕ A;
3   A = ((A - S[2i]) >>> B) ⊕ B;
4
5 B = B - S[1];
6 A = A - S[0];
```

TEA

El algoritmo *Tiny Encryption Algorithm* fue creado en 1994 por David Wheeler y Roger Needham con el objetivo de ser seguro, fácil de implementar y con un desempeño razonable (David J. Wheeler, 1994). El algoritmo es muy vérsatil ya que se puede implementar en una gran variedad de lenguajes de programación y no cuenta con tablas de reemplazo o expansión de llaves las cuales complican la implementación del algoritmo. Un ejemplo de esto es el algoritmo RC5 donde el paso de expansión de llave del algoritmo consume una gran cantidad de recursos y de tiempo de procesamiento.

Según David J. Wheeler (1994) este tipo de algoritmo puede reemplazar al DES el cual es más complejo de implementar (debido a que tiene tablas de sustitución y expansión de llave) con la particularidad de que se puede aumentar la seguridad aumentando la cantidad de rondas de cifrado donde difiere con el DES que cuenta con un número de rondas fijo (16).

Terminología del algoritmo

A continuación se definen las variables que se utilizan para describir el algoritmo:

- V (v0 y v1): Corresponde a un arreglo de 2 palabras de 32 bits que contiene el texto plano a cifrar.

- K (k0, k1, k2, k3): Corresponde a un arreglo de 4 palabras de 32 bits que conforman la llave de cifrado y descifrado.

- N: Corresponde al número de rondas para cifrar y descifrar.
- *delta*: Corresponde al valor $(\sqrt{5}-1)2^N$ que es un número derivado del *golden ratio* y se utiliza como parte del cifrado y descifrado.
- *sum,* $\{y,z\}$: Valor temporal de la sumatoria de delta y valores temporales del texto cifrado.

Pasos del algoritmo

El algoritmo al no contar con algún tipo de operación especial sobre la llave realiza únicamente los pasos de cifrado y descifrado, su implementación en C se muestra en los Listados 2.4 y 2.5 respectivamente.

Listado 2.4: Código en C del paso de cifrado en el algoritmo TEA (David J. Wheeler, 1994).

```
1  void code(long* v, long* k) {
2    unsigned long y=v[0],z=v[1], sum=0,
3
4    /* set up */
5    delta=0x9e3779b9,
6
7    /* a key schedule constant */
8    n=32;
9    while (n-->0) {
10     /* basic cycle start */
11     sum += delta;
12     y += ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]);
13     z += ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]);
14   }
15   /* end cycle */
16
17   v[0]=y; v[1]=z;
18 }
```

Listado 2.5: Código en C del paso de descifrado en el algoritmo TEA (David J. Wheeler, 1994)

```
1  void decode(long* v,long* k) {
2
3    unsigned long n=32, sum, y=v[0], z=v[1],
4    delta=0x9e3779b9;
5    sum=delta<<5;
6
7    /* start cycle */
8    while (n-->0) {
9      z-= ((y<<4)+k[2]) ^ (y+sum) ^ ((y>>5)+k[3]);
```

```
10      y-= ((z<<4)+k[0]) ^ (z+sum) ^ ((z>>5)+k[1]);
11      sum-=delta;
12    }
13    /* end cycle */
14
15    v[0]=y; v[1]=z;
16
17  }
```

3 Implementación de los algoritmos criptográficos

3.1 Implementación del algoritmo RC5

La implementación de este algoritmo se basó en el siguiente análisis:

- Se tienen parámetros variables y el objetivo del proyecto es poder analizar como la variación de los mismos impacta en la utilización de recursos del FPGA. Por tanto la manera más óptima de generar código en Verilog es que el código sea parametrizable.
- Se tiene arreglos de palabras los cuales son de tamaño variable de donde se leen y se escriben valores. La mejor forma de implementar esto es haciendo uso de memorias.
- Usualmente para cada paso del algoritmo se debe realizar lo siguiente:
  - Se lee un dato.
  - Se opera sobre el dato.
  - Se guarda el dato.
  - Se verifica si ya se terminaron de operar todo los datos.
  - Se cambia o mantiene la dirección para leer otro dato.

  Tomando esto en consideración se optó por implementar máquinas de estado finitas para poder segmentar fácilmente los pasos a seguir y que los mismos sean manejados mediante un reloj.
- Se tienen rotaciones hacia la izquierda y la derecha de N sobre M bits. Donde N es un valor calculado en tiempo de ejecución sobre una palabra de tamaño M. Esto se podría lograr mediante un barrel shifter que se genere dependiendo de los parámetros enviados a la hora de sintetizar, pero se consideró muy complicado y poco necesario. Por tanto se fijaron 3 tamaños de palabra comunes sobre los cuales se van a realizar los análisis: 16 bits, 32 bits y 64 bits. Esto facilita la escritura de código realizando un barrel shifter específico para cada tamaño de palabra.
- Se tienen sumas y restas de complemento a dos las cuales se ajustan perfectamente al uso de sumadores del FPGA.

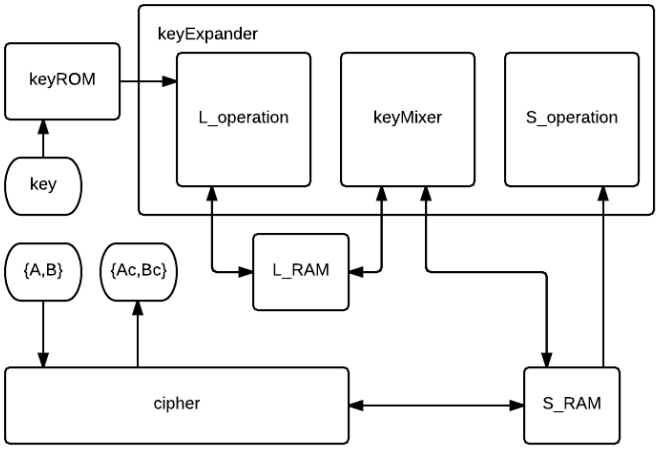

Figura 3.1: Diagrama de bloques del algoritmo de cifrado RC5.

- Se tiene XOR bit a bit que se ajustan perfectamente a una implementación en hardware.

A partir de esto podemos concluir que:

- Los arreglos L, S y el arreglo de K[0] a K[b-1] corresponden a memorias RAM.

- Cada uno de los *for* que muestran en el pseudo-código de los Listados 2.1, 2.2 y 2.3 corresponde a una máquina de estados la cual está controlada por un contador para manejar la cantidad de iteraciones.

Finalmente a partir de todos estos razonamientos se obtuvieron los diagramas de bloques para el cifrado y descifrado que se muestran en las Figuras 3.1 y 3.2 respectivamente.

Cada bloque representa un ROM o RAM donde esté indicado y los demás corresponden a máquinas de estados finitas. El diagrama de estados para los bloques *L_operation, S_operation* se muestra en la Figura 3.3. Note que ambos bloques comparten el diagrama de estados pero realizan diferentes operaciones. Para los bloques *keyMixer, cipher y decipher* los diagramas de estados se muestran en las Figuras 3.4, 3.5 y 3.6 respectivamente.

Un punto importante a aclarar es que el *keyROM* de las Figuras 3.1 y 3.2 se sintetiza en los resultados ya que por facilidad se optó por dejar el mismo como una entrada al circuito diseñado.

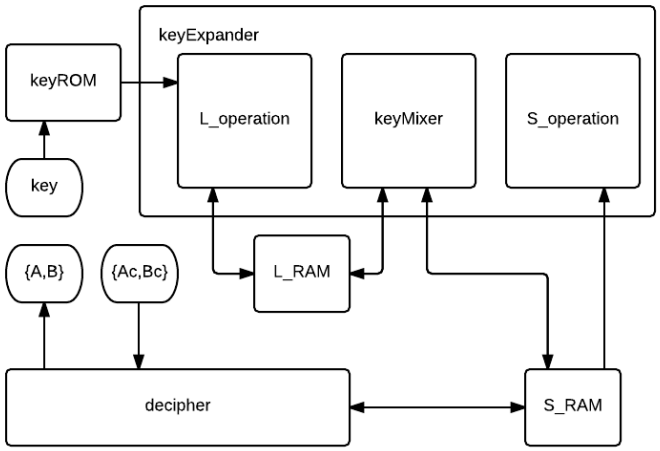

Figura 3.2: Diagrama de bloques del algoritmo de descifrado RC5.

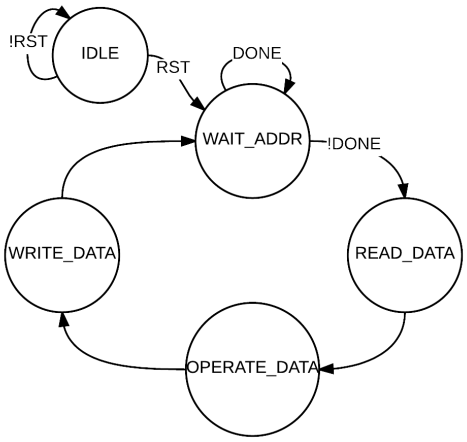

Figura 3.3: Diagrama de estados para los bloques *L_operation* y *S_operation* del algoritmo RC5.

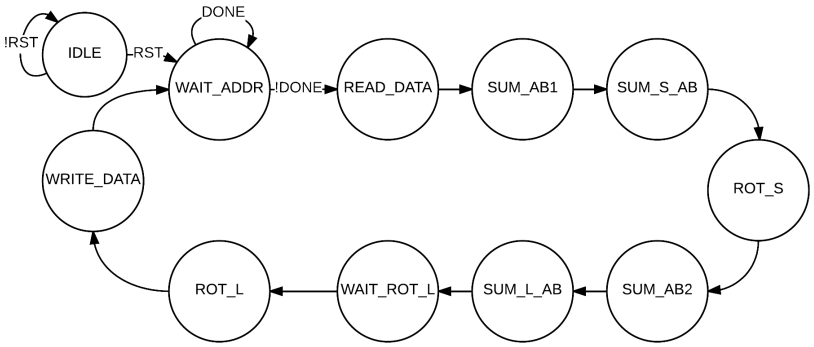

Figura 3.4: Diagrama de estados para el bloque *keyMixer* del algoritmo RC5.

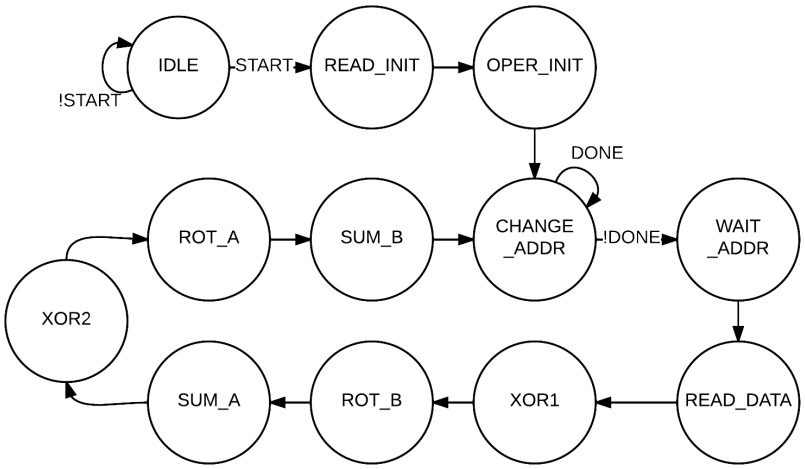

Figura 3.5: Diagrama de estados para el bloque *cipher* del algoritmo RC5.

3.2 Implementación del algoritmo TEA

La implementación del algoritmo se basó en el siguiente análisis:

- Nuevamente se tienen parámetros variables y el objetivo del proyecto es poder analizar como la variación de los mismos impacta en la utilización de recursos del FPGA. Por tanto la manera más óptima de generar código en Verilog es que el código sea parametrizable.

- Convertir la llave en una memoria RAM como se realizó en el algoritmo RC5 se consideró innecesario. Esto debido a que las operaciones no se

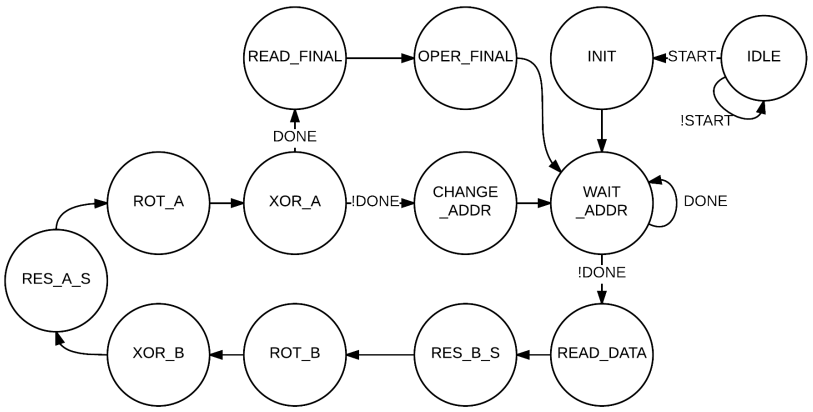

Figura 3.6: Diagrama de estados para el bloque *decipher* del algoritmo RC5.

realizan por bytes sino por palabras y solo se cuenta con 4 palabras que conforman la llave.

- Para que la frecuencia de reloj no fuera tan baja y con la meta a futuro de convertir el código en un *pipeline* se tomaron las operaciones dentro de los *while* de los Listados 2.4 y 2.5 y se segmentaron para que cada ciclo de reloj solo se realice una suma o XOR.

- Al estar basado en una red Feistel debe ser posible variar el número de rondas. El código en C de los algoritmos en el artículo (David J. Wheeler, 1994) no contaba con la variación del número de rondas y se tuvo que analizar que parámetros se debían variar para lograrlo. Según David J. Wheeler (1994) se debe alterar el valor de *sum* de algoritmo de descifrado (Listado 2.5) y convertirlo en el valor de *delta* multiplicado por la cantidad de rondas.

- El algoritmo se encuentra orientado a trabajar sobre texto plano de 64 bits (2 palabras de 32 bits, v0 y v1) pero se adaptó para que el algoritmo trabajara sobre palabras de tamaño variable así como de tamaño de llave variable, que depende directamente del tamaño de la palabra. La finalidad de este cambio es poder realizar el análisis variando los diferentes parámetros.

A partir de estos análisis se puede concluir que:

- La llave completa se puso como entrada al circuito.

- La forma más óptima (obviando el pipeline) para describir los algoritmos de cifrado y descifrado es mediante máquinas de estados finitas.

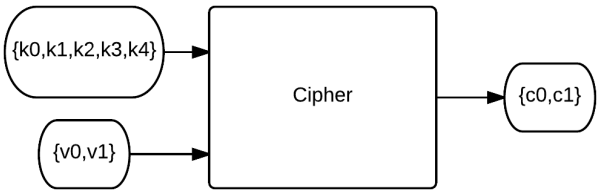

Figura 3.7: Diagrama de bloques del algoritmo de cifrado TEA.

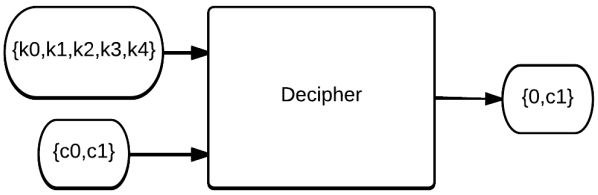

Figura 3.8: Diagrama de bloques del algoritmo de descifrado TEA.

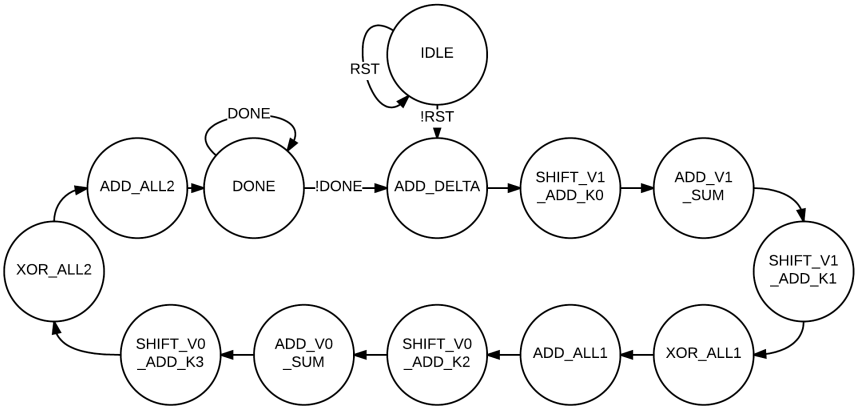

Figura 3.9: Diagrama de estados para el bloque *cipher* del algoritmo TEA.

Con todo lo descrito anteriormente se crearon los diagramas de bloques que se muestran en las Figuras 3.7 y 3.8 para los algoritmos de cifrado y descifrado respectivamente. Además los bloques *cipher* y *decipher* de esas Figuras corresponden a máquinas de estados finitas las cuales tienen los diagramas de estados que se muestran en la Figuras 3.9 y 3.10 respectivamente.

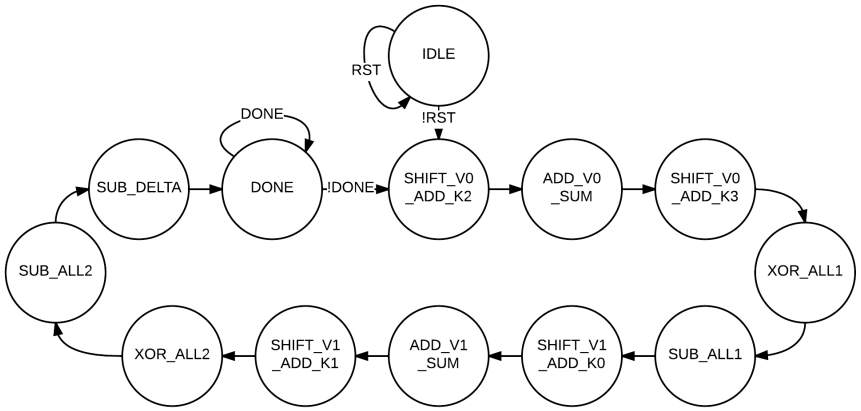

Figura 3.10: Diagrama de estados para el bloque *decipher* del algoritmo TEA.

4 Resultados y análisis

En este capítulo se van a mostrar y analizar los resultados de la implementación de los algoritmos criptográficos RC5 y TEA. Primero se presentan resultados y análisis individuales, para posteriomente realizar el análisis comparativo.

Las síntesis e implementación de los algoritmos en el FPGA se llevó a cabo haciendo uso de la herramienta de desarrollo *Vivado Design Suite* de *Xillinx* bajo la licencia gratuita *WEBPack*. Se eligió como objetivo de implementación la tarjeta xc7k70tfbv676-1 de la familia de FPGAs *Kintex-7*. En la Tabla 4.1 se muestran todos los recursos disponibles de la misma.

Tabla 4.1: Recursos disponible para el FPGA Kintex-7 xc7k70tfbv676-1 de Xillinx.

Componente	Cantidad Disponible
Slices	10250
Slice LUTs	41000
Slice Registers	82000
Block RAM	135
DSPs	240
I/O ports	300

Para obtener las métricas de análisis se utilizaron los reportes de temporización y utilización que la herramienta ofrece al finalizar el proceso de implementación. Se guardaron estos archivos después de cada sintesís e implementación y se escribió un *script* de Python para extraer los datos deseados y escribirlos a un archivo CSV (*Comma Separated Values*). Luego de esto se utilizó *Excel* para graficar las diferentes métricas haciendo uso de tablas pivote para un análisis más eficiente.

Cabe aclarar que aunque en un principio no se iba a tomar en cuenta el análisis de temporización, específicamente la frecuencia máxima de reloj, la misma fue agregada a la lista de métricas a analizar debido a que en la implementación realizada la frecuencia máxima de reloj afecta directamente el tiempo de procesamiento para cifrar y descifrar datos y por tanto es una métrica muy importante a analizar. Adicionalmente se puede destacar el hecho de que igualmente a causa de la implementación, el variar la cantidad de rondas que cifran o descifran los datos no afectan de ninguna manera la cantidad de

recursos del FPGA y solo afectan el tiempo de procesamiento total de cifrado o descifrado.

Otro aspecto importante es que se asumió que los módulos *top* de los algoritmos son módulos que serán parte de un dispositivo más grande (por ejemplo un procesador) y por tanto que ese dispositivo hará uso de tanto los módulos de cifrado y descifrado, así los módulos serán sintetizados e implementados simúltaneamente para cada algoritmo, ya que el análisis de ambos módulos tendrá más valor que el de los módulos por separado. Particularmente para el algoritmo RC5 se hizo que los módulos de cifrado y descifrado compartan el módulo que se encarga de la expansión de llave (*keyExpander* en las Figuras 3.1 y 3.2) esto con la finalidad de consumir la menor cantidad de recursos y hacer lo más óptimo posible el diseño.

Un problema al que se enfrentó al implementar los algoritmos en el FPGA es que la cantidad de salidas y entradas del diseño era muy grande para que el mismo fuera implementable en el FPGA. La solución a esto corresponde a un módulo para serializar y deserializar las entradas y salidas pero el mismo no fue implementado ya que no era uno de los objetivos del proyecto, además de que se consideró innecesario ya que, como se indicó anteriormente, estos algoritmos formarían parte de un dispositivo más grande que ya contará con este módulo. Así para poder implementar el diseño simplemente se pusieron dos puertos, uno de entrada y otro de salida, ambos del tamaño de la palabra y las señales de control de los módulos.

También se recalca el hecho de que en ninguno de los algoritmos analizados el FPGA hizo uso de los bloques de DSP con los que cuenta, por tanto esta métrica fue descartada de los análisis y no se hará mención alguna de la misma de aquí en adelante.

4.1 Resultados y análisis del algoritmo TEA

Inicialmente se corroboró que el algoritmo funcionara correctamente, esto se logró comparando los resultados de texto cifrado y descifrado con el código en C que se encuentra en los Listados 2.4 y 2.5[1]. La Tabla 4.2 muestra el resultado de cifrado y descifrado al simular con diferentes tamaños de palabra para 32 rondas de cifrado/descifrado. Además la tabla en la columna *layout* indica un número de figura al cual está ligado la implementación para el tamaño de palabra. Este *layout* corresponde al dado por la herramienta de desarrollo para observar la distribución de recursos en el circuito. Además el camino resaltado en blanco en cada *layout* corresponde al *critical path* de cada implementación.

[1]El código para verificar fue ligeramente diferente al de los listados debido a que se tuvo que ajustar para poder variar el tamaño de palabra.

Tabla 4.2: Resultados de simulación del TEA para diferentes tamaños de palabra.

Tamaño de palabra (bits)	Texto plano (hex)	Texto cifrado (hex)	Figura *layout*
8	0d8d	aba7	4.1
16	7b0d998d	829789bc	4.2
32	3d45f7a7235fcb21	f5509056d5db9e6a	4.3
64	0000000006b97b0d 0000000046df998d	0d85f88a9ef595ff 4810210e9e2a4f5e	4.4
128	3ca67c8e15890877 6dcc3a7b41cb88e6 9a34483d3b1a68a4 1235130bf207ee95	c0144f6419ce0203 dbee17be33c2bf5c 87b4dc12f9cbe22f f2f826efc585fa46	4.5

En el caso de este algoritmo los parámetros que se variaron y cómo se variaron se muestran en la Tabla 4.3 donde se fijó la cantidad de rondas en el valor nominal del algoritmo, 32 rondas de cifrado/descifrado[2]. Además en esa misma tabla se muestra el consumo de recursos del FPGA al variar esos parámetros. Note en los Listados 2.4 y 2.5 como por el algoritmo no es posible variar independientemente el tamaño de la llave del tamaño de la palabra a cifrar y por tanto no es posible realizar un análisis independiente del análisis de la variación del tamaño de llave y como afecta los recursos del FPGA.

De la Tabla 4.3 se pueden extraer las gráficas que se muestran en las Figuras 4.6, 4.7 y 4.8.

La Figura 4.6 muestra como varía la cantidad de *slices* tanto de tipo L como M conforme el tamaño de palabra varía. La Figura 4.7 muestra como varía la cantidad de *slice registers* y LUTs al variar el tamaño de la palabra. Note como después de variar la palabra de 32 bits a 64 bits la cantidad de recursos comienza a aumentar mostrando un comportamiento exponencial. Además la relación de ambas gráficas es la esperada ya que muestran la misma tendencia debido a que los *slices* se encuentran compuestos por registros y LUTs.

La Figura 4.8 muestra la variación de la frecuencia máxima de reloj conforme el tamaño de palabra varía. El comportamiento es el esperado, ya que la frecuencia de reloj disminuye mostrando un comportamiento lineal decrecien-

[2]Para tamaños de palabra mayores a 128 se consideró innecesario el análisis debido a que ya las gráficas mostraban un comportamiento y los tiempos de síntesis e implementación en el FPGA se estaban volviendo muy extensos.

Tabla 4.3: Variación de parámetros del algoritmo TEA y consumo de recursos del FPGA.

Tamaño palabra (bits)	Tamaño llave (bits)	LUT as Logic	Slice L	Slice M	Slice Registers	Frecuencia máx. (MHz)
8	32	217	45	27	176	434,783
16	64	383	74	38	328	392,157
32	128	737	133	90	632	338,983
64	256	1439	243	179	1240	303,030
128	512	2841	439	357	2456	217,391

te. Esto puede deberse a que la herramienta de desarrollo optimiza el proceso de *place & route* para lograr la mejor respuesta del diseño.

Finalmente los datos de la Tabla 4.3 fueron normalizados respecto al valor mínimo de cada columna (por ejemplo 217 en la columna *LUT as Logic*) con la finalidad de dar un análisis más acertado sobre cual sería el tamaño de palabra más conveniente a utilizar para este algoritmo. Los datos normalizados se muestran en la Tabla 4.4 y se reflejan en la gráfica de la Figura 4.9.

Tabla 4.4: Variación de parámetros del algoritmo TEA y consumo de recursos del FPGA normalizados.

Tamaño palabra (bits)	Tamaño llave (bits)	LUT as Logic	Slice L	Slice M	Slice Registers	Frecuencia máx. reloj
8	32	1,00	1,00	1,00	1,00	2,00
16	64	1,76	1,64	1,41	1,86	1,80
32	128	3,40	2,96	3,33	3,59	1,56
64	256	6,63	5,40	6,63	7,05	1,39
128	512	13,09	9,76	13,22	13,95	1,00

Note como en esta gráfica el análisis se puede realizar de manera global debido a que los datos se muestran normalizados. Además se puede notar que el comportamiento es el esperado ya que se observa como conforme se varía el tamaño de palabra los recursos consumidos por el FPGA tienden a aumentar con un comportamiento similar entre si. En el caso de la frecuencia máxima de reloj se observa como va disminuyendo conforme se varía el tamaño de palabra pero tiene un comportamiento menos abrupto, como es de esperar por su tendencia lineal y optimizaciones de la herramienta.

Finalmente bajo los análisis dados anteriormente se puede concluir que el tamaño de palabra ideal podría ser cualquiera menor a 32 bits. Las razones de esto son las siguientes:

- El tamaño de palabra de 32 bits, implica una entrada de texto plano de 64 bits, la cual se ajusta perfectamente al tamaño de palabra actual de la mayoría de procesadores de computadoras personales. También existen muchos microcontroladores con procesadores con tamaños de palabra de 32 bits o 16 bits, lo cual indicaría para el algoritmo tamaños de palabra de 8 bits o 16 bits. El beneficio de "acoplar" de alguna manera las entradas y salidas del algoritmo a las entradas y salidas del procesador es que en el cifrado puede procesar inmediatamente la información conforme esta sale del procesador y en el caso del descifrado es que puede entregar inmediamente al procesador la información descifrada y continuar descifrando datos.[3]

- Las frecuencias máximas de reloj a las que trabajan esas implementaciones son mucho mayores que en las implementaciones de palabras de mayor tamaño.

- Aunque puede ocurrir una disminución en el nivel de seguridad del algoritmo al trabajar con palabras más pequeñas, esta se puede aumentar aumentando la cantidad de rondas de cifrado y al trabajar a una frecuencia más alta el impacto en el desempeño no es tanto.

- Se ahorra una gran cantidad de recursos en el FPGA el cual se traduce directamente en costos de producción.

[3]Inclusive existen microcontroladores con procesadores de 8 bits por ejemplo el HC 08 de Freescale (`http://www.freescale.com/products/more-processors/8-bit-mcus/hc08:HC08FAMILY?cof=0&am=0`). Entonces sería válido realizar un análisis para 4 bits de tamaño de palabra del algoritmo.

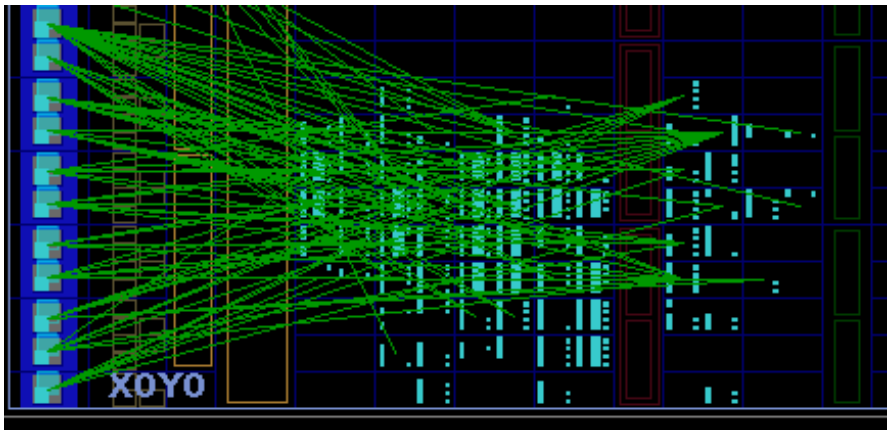

Figura 4.1: *Layout* de la implementación del algoritmo TEA de 8 bits de tamaño de palabra y 32 rondas de cifrado/descifrado.

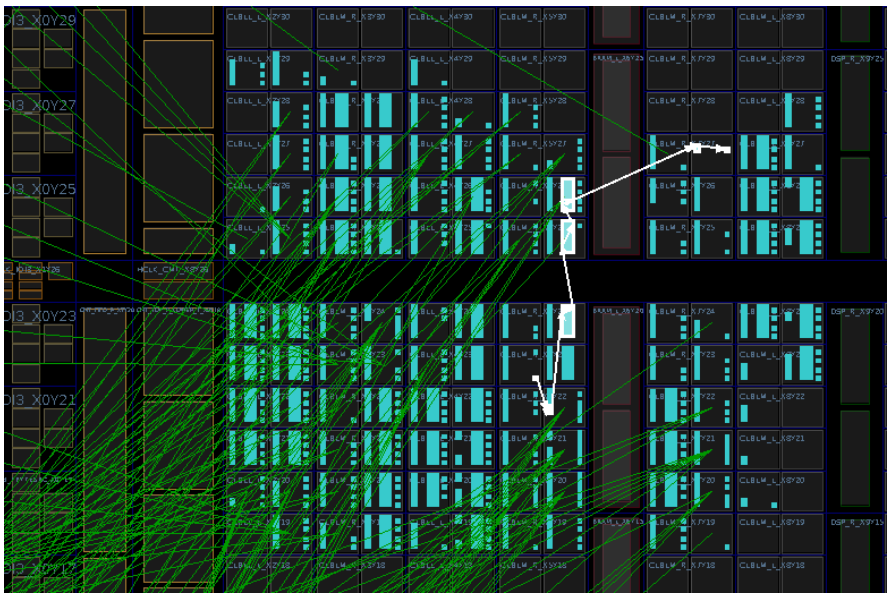

Figura 4.2: *Layout* de la implementación del algoritmo TEA de 16 bits de tamaño de palabra y 32 rondas de cifrado/descifrado.

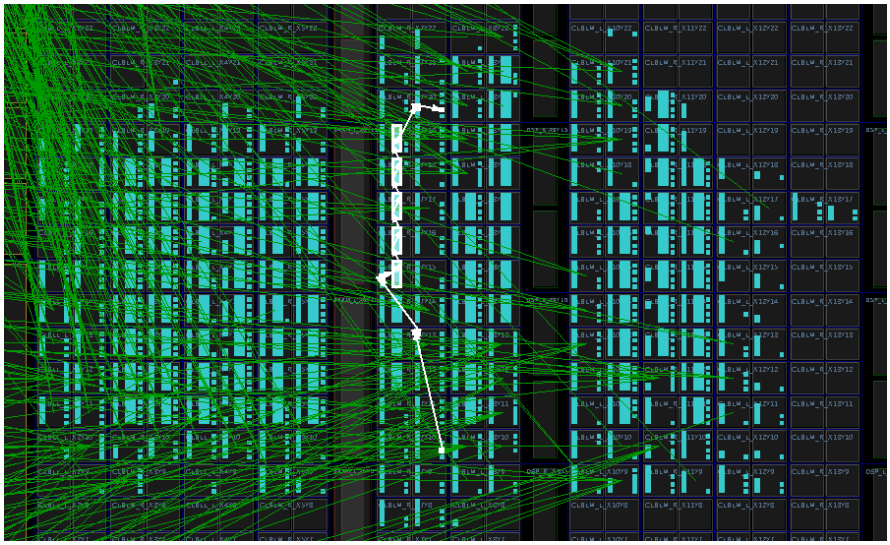

Figura 4.3: *Layout* de la implementación del algoritmo TEA de 32 bits de tamaño de palabra y 32 rondas de cifrado/descifrado.

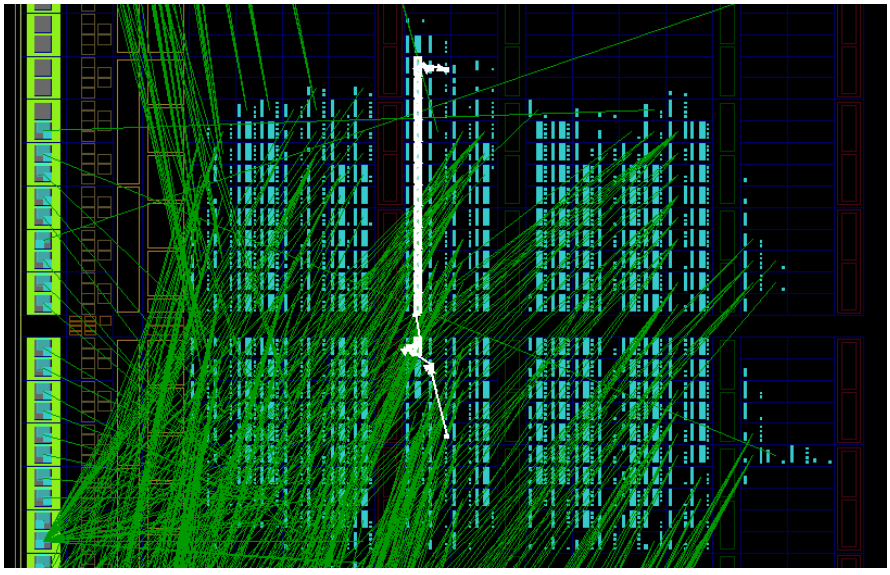

Figura 4.4: *Layout* de la implementación del algoritmo TEA de 64 bits de tamaño de palabra y 32 rondas de cifrado/descifrado.

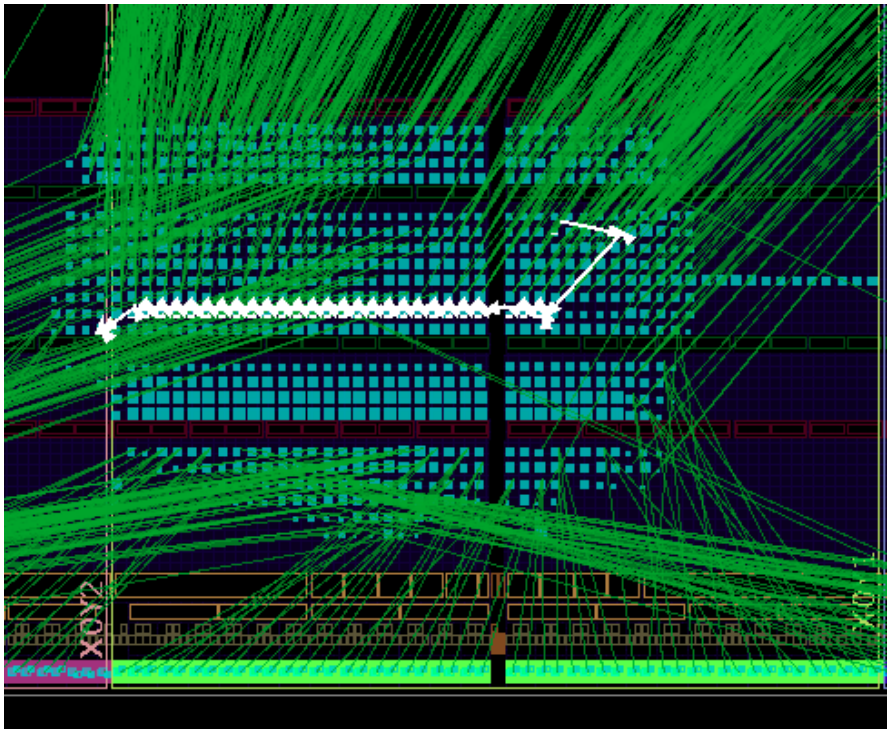

Figura 4.5: *Layout* de la implementación del algoritmo TEA de 128 bits de tamaño de palabra y 32 rondas de cifrado/descifrado.

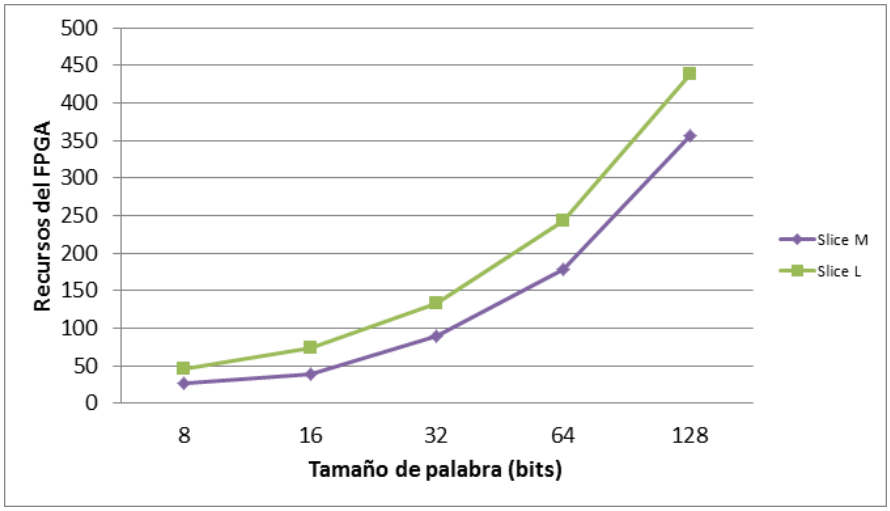

Figura 4.6: Gráfica de variación de *slice L* y *slice M* contra la variación del tamaño de palabra en bits al implementar el algoritmo TEA.

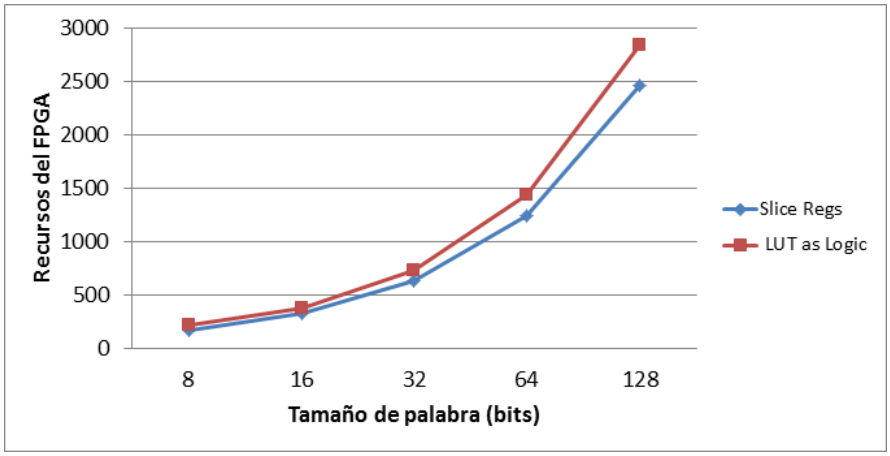

Figura 4.7: Gráfica de variación de *slice registers* y LUTs contra la variación del tamaño de palabra en bits al implementar el algoritmo TEA.

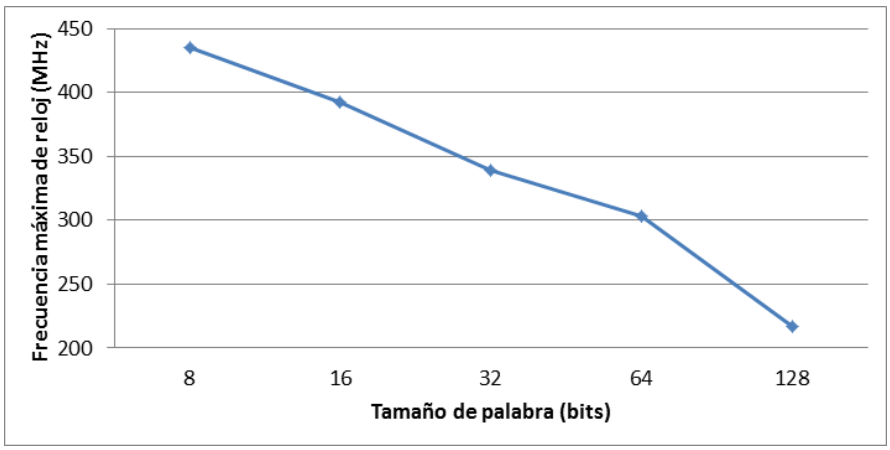

Figura 4.8: Gráfica de variación de la frecuencia máxima de reloj contra la variación del tamaño de palabra en bits al implementar el algoritmo TEA.

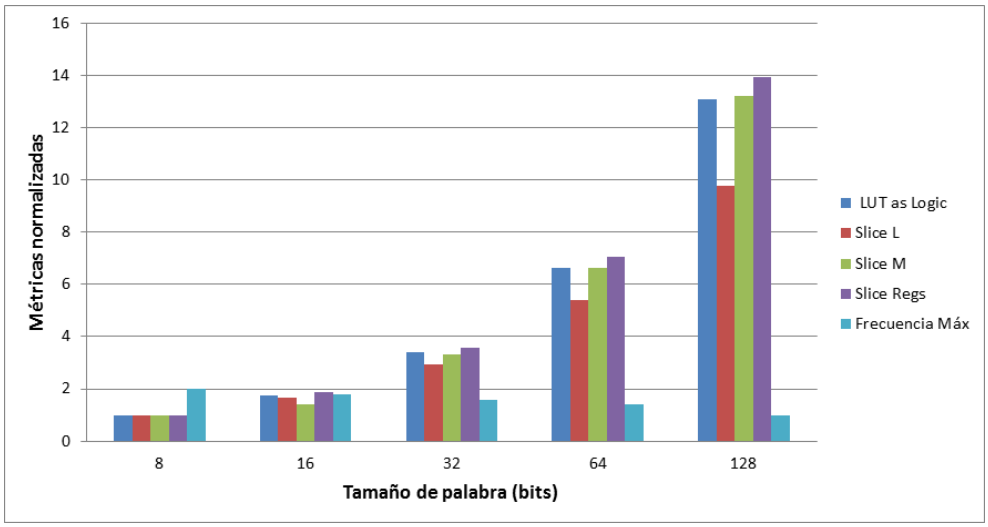

Figura 4.9: Gráfica de consumo de recursos y frecuencia máxima (normalizados respecto al valor mínimo) del FPGA contra la variación del tamaño de palabra en bits al implementar el algoritmo TEA.

4.2 Resultados y análisis del algoritmo RC5

Inicialmente se corroboró que el algoritmo funcionara de forma correcta. Esto se llevó a cabo mediante la comparación de los resultados de texto cifrado y descifrado con el código en C que se encuentra en el artículo del algoritmo: (Rivest, 1997). La Tabla 4.5 muestra el resultado de cifrado y descifrado al simular el algoritmo variando el tamaño de palabra y la Tabla 4.6 muestra los resultados de simular el algoritmo variando el tamaño de llave. Al igual que el algoritmo anterior, la tabla en la columna *layout* indica un número de figura al cual se encuentra ligada la implementación.

Tabla 4.5: Resultados de simulación para distintos tipos del RC5 variando el tamaño de palabra con una llave de 16 bytes igual a $91CEA91001A5556351B241BE19465F91_{hex}$.

Tipo RC5	Texto plano (hex)	Texto cifrado (hex)	Figura *layout*
16/12/16	a5214b15	a496d848	4.10
32/12/16	eedba521 6d8f4b15	ac13c0f7 52892b5b	4.11
64/12/16	00000000 eedba521 00000000 6d8f4b15	829c9641 f96ead46 f91c9890 738c8807	4.12

Posterior a verificar el funcionamiento del algoritmo se procedió a sintetizar e implementar el algoritmo en el FPGA (con las variaciones descritas en las Tablas 4.5 y 4.6) haciendo uso del *Vivado Design Suite*. Con esto se obtuvo la variación de recursos del FPGA (mediante el mismo procedimiento descrito en la Sección 4.1) mostrada en las Tablas 4.7 y 4.8. En el caso de las variaciones de palabra, los valores fueron elegidos debido a que el autor del algoritmo indica que la implementación nominal del algoritmo es la RC5-32/12/16. Por tanto se quería observar la variación en el consumo de recursos alrededor del tamaño de palabra nominal y eligiendo valores típicos de tamaños de palabra (potencias de 2). En el caso del tamaño de llave los tamaños fueron elegidos con el fin de poder realizar el análisis comparativo más adelante.

Resultados y análisis de la variación del tamaño de la llave

De la Tabla 4.7 se generó la gráfica mostrada en la Figura 4.16, la cual muestra la variación del consumo de recursos del FPGA y la frecuencia máxima de

Tabla 4.6: Resultados de simulación para distintos tipos del RC5 variando el tamaño de llave.

Tipo RC5	Llave (hex)	Texto plano (hex)	Texto cifrado (hex)	Figura *layout*
32/16/7	2c5b2aa c54b1a5	12153524 c0895e81	9e6edba6 9b789a70	4.13
32/16/12	5e68d4 564a2c 5b2aac 54b1a5	f232b52 aeeebba13	a4db4976 94726be1	4.14
32/16/16	d4543e13 5e68d456 4a2c5b2a ac54b1a5	eedba521 6d8f4b15	b6c95c5c e51616b8	4.15

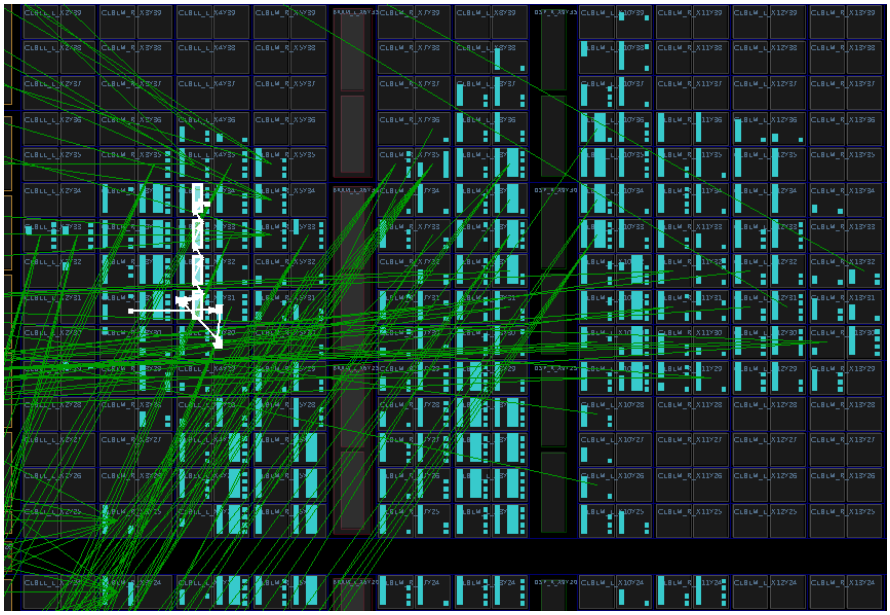

Figura 4.10: *Layout* de la implementación del algoritmo RC5-16/12/16.

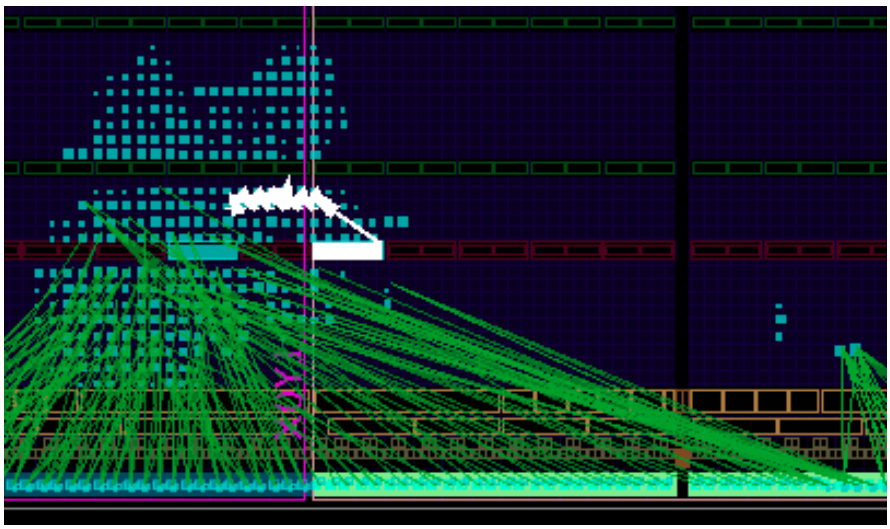

Figura 4.11: *Layout* de la implementación del algoritmo RC5-32/12/16.

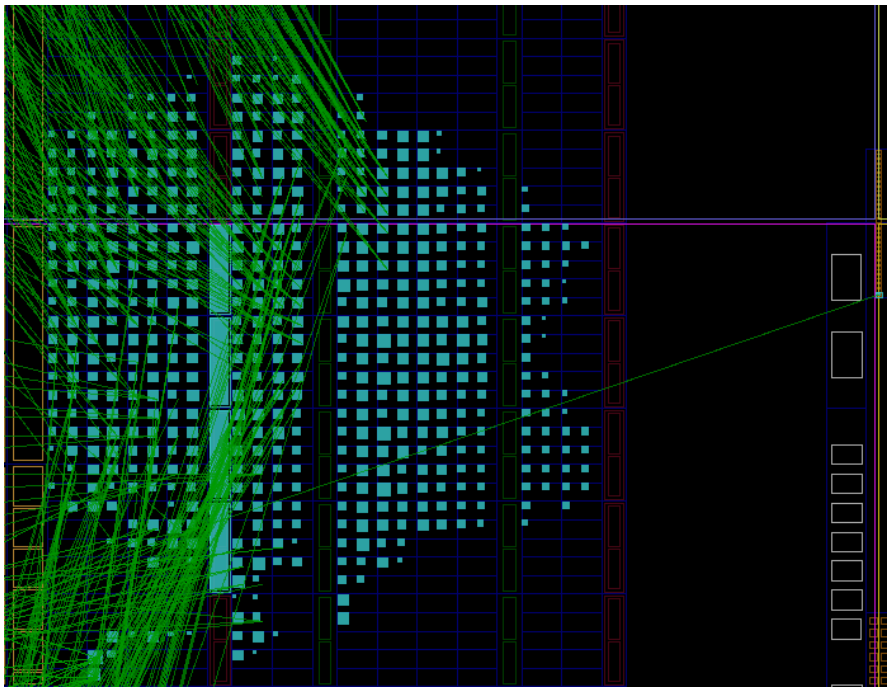

Figura 4.12: *Layout* de la implementación del algoritmo RC5-64/12/16.

Tamaño llave (bytes)	LUT as Logic	RAM	LUT as Memory	Slice L	Slice M	Slice Regs	Frec. máx. (MHz)
7	993	2	8	180	135	685	227,27
12	993	2	8	194	127	688	217,39
16	991	2	8	187	129	688	212,77

Tabla 4.7: Variación del tamaño de la llave y recursos del FPGA para el algoritmo RC5-32/16/B

Tamaño palabra (bits)	LUT as Logic	RAM	LUT as Memory	Slice L	Slice M	Slice Regs	Frec. máx. (MHz)
16	578	0	40	112	82	446	377,36
32	972	2	8	188	139	687	212,77
64	1911	4	8	314	236	1273	192,31

Tabla 4.8: Variación del tamaño de la palabra y recursos del FPGA para el algoritmo RC5-W/12/16

reloj conforme varía el tamaño de llave. El comportamiento mostrado fue casi constante demostrando que la variación del tamaño de la llave hasta de un tamaño mayor al doble, afecta de manera mínima el consumo de recursos del FPGA. Esto es muy valioso debido a que al aumentar el tamaño de llave se aumenta la seguridad del algoritmo pero no afecta el consumo de recursos del FPGA. El aspecto que si se ve afectado con esta variación es el tiempo de procesamiento, ya que se debe leer y escribir más veces a la memoria que resguarda la llave. Así se podría concluir del análisis, que el tamaño de llave ideal depende de cuanta seguridad se quiere y que tan rápido se necesita llevar a cabo el proceso de cifrado/descifrado de los datos.

Resultados y análisis de la variación del tamaño de la palabra

De la Tabla 4.8 se extrajeron las gráficas de la Figuras 4.17, 4.18, 4.19 y 4.20.

La Figura 4.17 muestra la variación de la cantidad de *Slices L* y *Slices M* consumidos conforme se aumenta el tamaño de palabra. Además la Figura 4.18 muestra la variación de la cantidad de *Slice Registers* y LUTs conforme se aumenta el tamaño de palabra. Como se puede observar de ambas gráficas el consumo de recursos del FPGA muestra un comportamiento exponencial. Además es destacable que ambas figura muestran el mismo comportamiento

ya que, como se mencionó en el análisis del algoritmo TEA, los *Slices* se componen de registros y LUTs.

La Figura 4.19 muestra el consumo de *LUT as memory* y bloques de memoria RAM al variar el tamaño de palabra. Se observa que, la implementación del algoritmo con tamaño de palabra de 16 bits no hace uso de ningún bloque de memoria RAM y más bien utiliza LUTs como memoria. Las otras dos implementaciones hacen uso de bloques de memoria RAM causando el desuso de gran parte de los LUTs como memoria, disminuyendo esta métrica de manera drástica y convirtiendola en un valor constante. También de la gráfica se puede observar el comportamiento lineal del consumo de bloques de memoria RAM. Esto se ajusta al comportamiento esperado de que al duplicar el tamaño de palabra se vaya duplicando el uso de bloques de memoria RAM.

La Figura 4.20 muestra la variación de la frecuencia máxima de reloj al variar el tamaño de palabra. Como se puede observar para la implementación con tamaño de palabra de 16 bits la frecuencia es mucho más alta que para las otras dos implementaciones. Esto se debe al cambio de utilizar LUTs como memoria a utilizar bloques de memoria RAM, los cuales son mucho más lentos. Además se nota como al pasar de 32 bits de tamaño de palabra a 64 bits, la variación de frecuencia máxima es pequeña, ya que al no mapear la memoria a LUTs sino a bloques de memoria RAM no se tiene una memoria distribuida sino centralizada, produciendo que la frecuencia sea mucho más estable.

Finalmente y al igual que con el algoritmo TEA se normalizaron los datos para tener una visión global para analizar. Los datos normalizados se muestran en la Tabla 4.9 y se reflejan en la gráfica de la Figura 4.21. Se exceptúo del análisis normalizado las métricas *LUT as memory* y RAM debido a que RAM no era un parámetro normalizable (para el tamaño de palabra de 16 bits tiene un valor de cero) y que *LUT as memory* varió de forma inconsistente por el uso de bloques de memoria RAM, agregando poco valor al análisis. De la gráfica el comportamiento es el esperado ya que se observa como conforme se varía el tamaño de palabra los recursos consumidos por el FPGA tienden a aumentar con un comportamiento similar entre si. En el caso de la frecuencia máxima de reloj se observa como va disminuyendo conforme se varía el tamaño de palabra pero tiene un comportamiento más estable posterior al uso de bloques de memoria RAM.

Tamaño palabra (bits)	LUT as Logic	Slice L	Slice M	Slice Regs	Frec. máx. (MHz)
16	1	1	1	1	1,96
32	1,68	1,68	1,70	1,54	1,11
64	3,31	2,80	2,88	2,85	1

Tabla 4.9: Variación del tamaño de la palabra y recursos del FPGA normalizados para el algoritmo RC5-W/12/16.

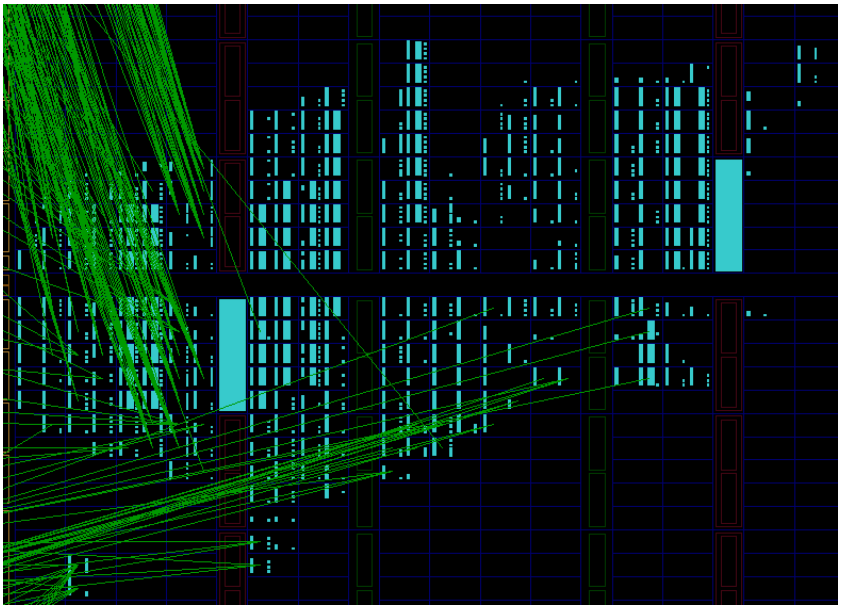

Figura 4.13: *Layout* de la implementación del algoritmo RC5-32/16/7.

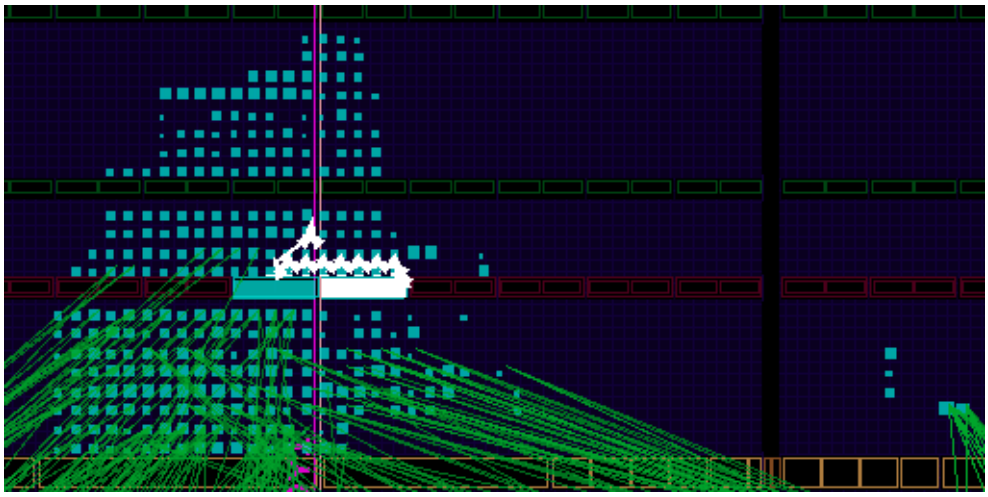

Figura 4.14: *Layout* de la implementación del algoritmo RC5-32/16/12.

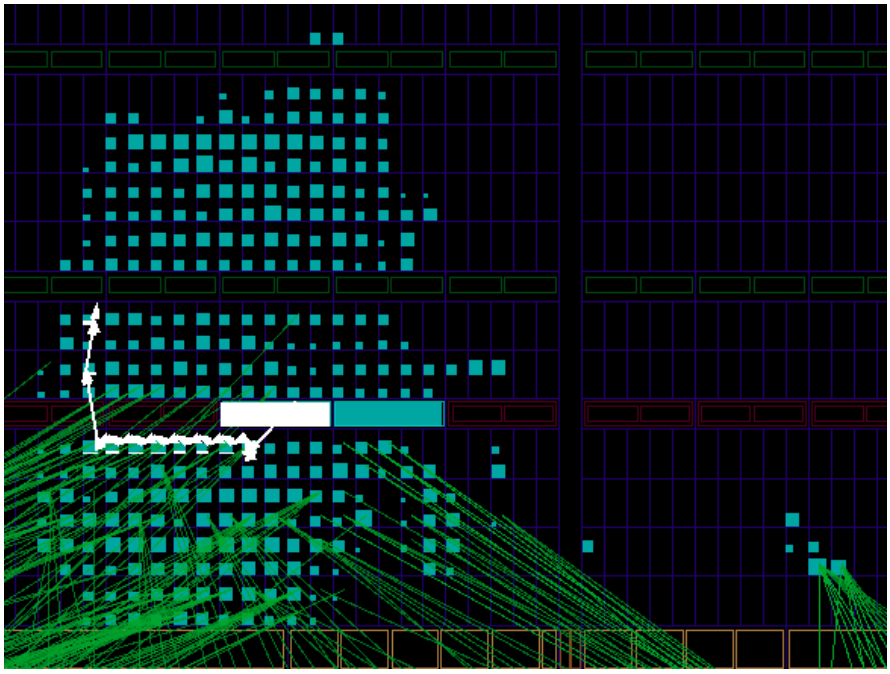

Figura 4.15: *Layout* de la implementación del algoritmo RC5-32/16/16.

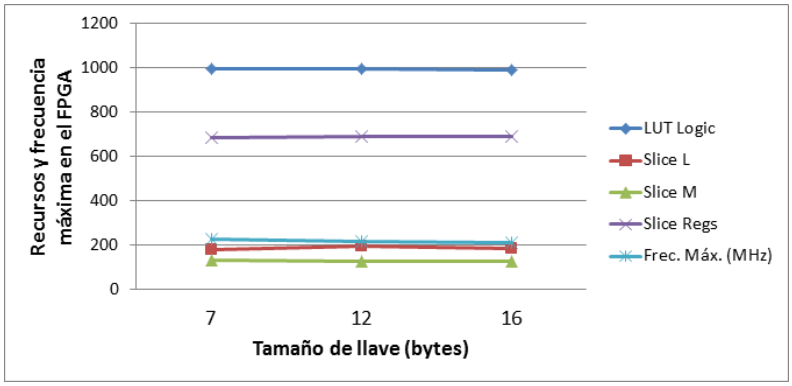

Figura 4.16: Gráfica de consumo de recursos y frecuencia máxima del FPGA contra la variación del tamaño de llave en bytes al implementar el algoritmo RC5.

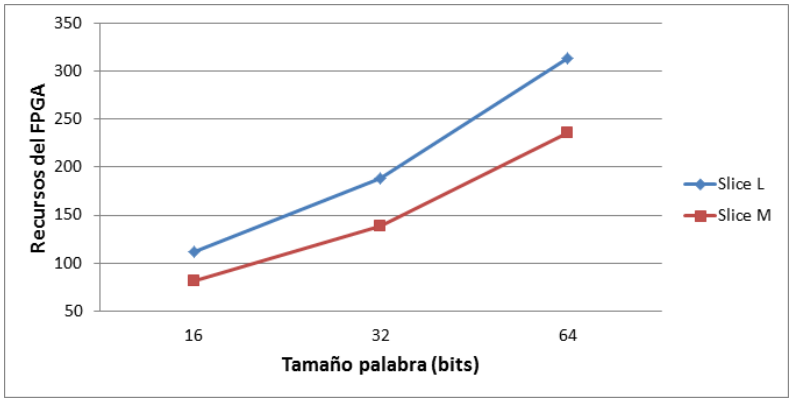

Figura 4.17: Gráfica de consumo de *Slices L* y *Slices M* del FPGA contra la variación del tamaño de la palabra al implementar el algoritmo RC5.

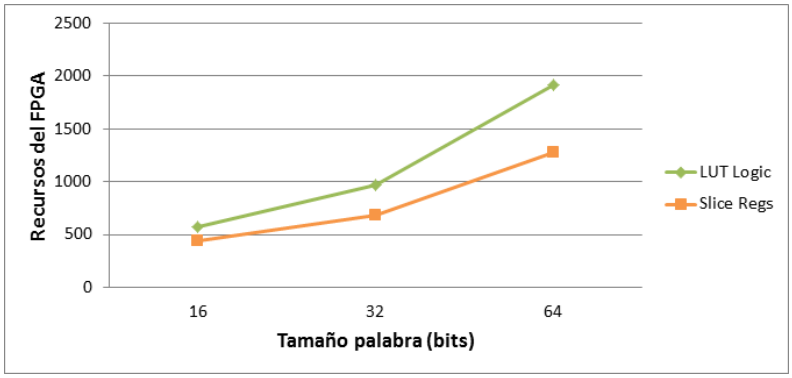

Figura 4.18: Gráfica de consumo de *Slice Registers* y LUTs del FPGA contra la variación del tamaño de la palabra al implementar el algoritmo RC5.

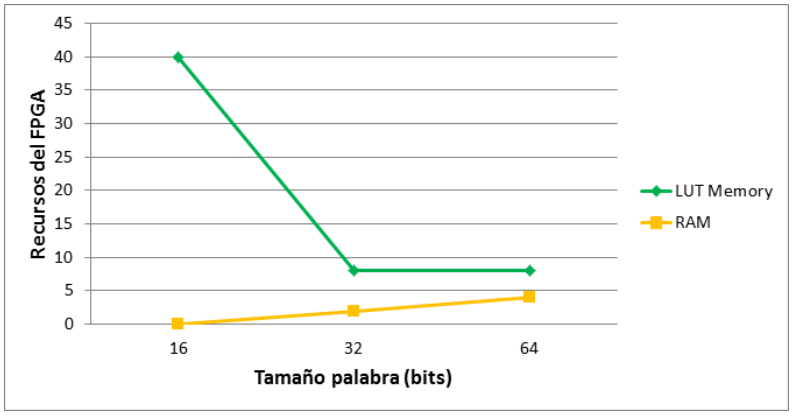

Figura 4.19: Gráfica de consumo de *LUT as memory* y bloques de memoria RAM del FPGA contra la variación del tamaño de la palabra al implementar el algoritmo RC5.

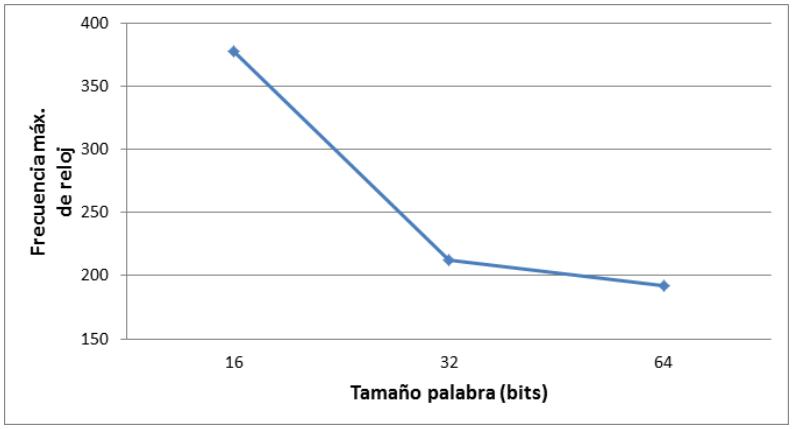

Figura 4.20: Gráfica de frecuencia máxima del FPGA contra la variación del tamaño de la palabra al implementar el algoritmo RC5.

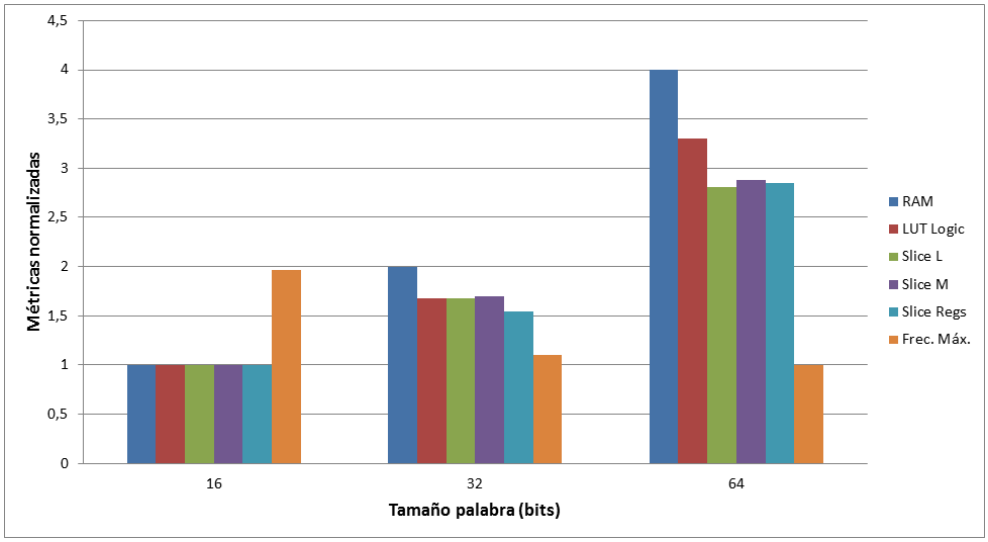

Figura 4.21: Gráfica de consumo de recursos y frecuencia máxima (normalizados respecto al valor mínimo) del FPGA contra la variación del tamaño de palabra al implementar el algoritmo RC5.

4.3 Análisis comparativo

Con la finalidad de realizar un análisis comparativo justo es necesario tener un punto de referencia para la escogencia de los parámetros de cada algoritmo. Esto se logró tomando como referencia lo descrito en los artículos de cada algoritmo. En ambos artículos se hace una referencia a cual implementación del algoritmo es "equivalente" a utilizar el algoritmo DES. Rivest (1997) indica que:

> *"Como un ejemplo, considere el problema de reemplazar DES con un "equivalente" del algoritmo RC5. Una elección razonable sería RC5-32/16/7 como reemplazo."*

Así como David J. Wheeler (1994) indica:

> *"Este tipo de algoritmo puede reemplazar al DES en software, y es lo suficientemente corto para ser escrito en cualquier programa o computadora."*

Por tanto el RC5-32/16/7 y el TEA con 32 rondas y tamaño de palabra de 32 bits fueron elegidos para realizar el análisis comparativo. Los parámetros y recursos consumidos de cada implementación del algoritmo se muestran en la Tabla 4.10. Nuevamente se realizó una normalización de los datos como se muestra en la Tabla 4.11 y se graficaron los datos de la misma en la Figura 4.22.

Algoritmo	Tamaño palabra (bits)	Tamaño llave (bytes)	Número rondas	LUT as Logic	RAM	LUT as Memory	Slice L	Slice M	Slice Registers	Frec. máx. (MHz)
RC5	32	7	16	993	2	8	180	135	685	227,27
TEA	32	16	32	737	0	0	133	90	632	338,983

Tabla 4.10: Tabla comparativa de parámetros y consumo de recursos de los algoritmos RC5 y TEA.

De la Figura 4.22 se puede observar claramente como el algoritmo TEA tiene mejores métricas en todo aspecto, consume menos recursos y tiene una frecuencia máxima mucho mayor. Esto se puede deber a varias razones:

- El algoritmo TEA no cuenta con la variación independiente de la llave y por tanto el algoritmo no necesita ser tan complicado, disminuyendo la cantidad de lógica necesaria para implementarlo.

Algoritmo	LUT as Logic	Slice L	Slice M	Slice Registers	Frec. máx. (MHz)
TEA	1	1	1	1	1,49
RC5	1,35	1,35	1,5	1,08	1

Tabla 4.11: Tabla comparativa normalizada (respecto al valor mínimo) del consumo de recursos y frecuencia máxima de los algoritmos RC5 y TEA.

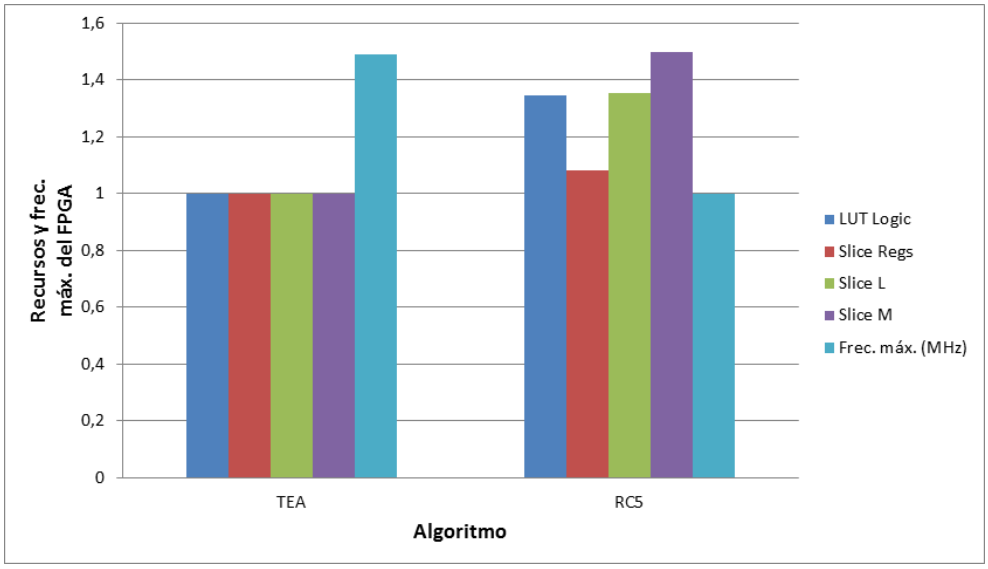

Figura 4.22: Gráfica del consumo de recursos y frecuencia máxima (normalizados respecto al valor mínimo) del FPGA para los algoritmos algoritmo RC5 y TEA.

- El algoritmo RC5 hace uso de bloques de memoria RAM, el TEA no lo hace. Esto produce que la cantidad de lógica aumente y además se vuelva más lenta, al tener que leer y escribir a varias memorias.

Otro aspecto que debemos tomar en cuenta es el tiempo que toma el proceso de cifrado y descifrado. En el RC5 es mucho mayor que en el TEA, tanto en frecuencia de reloj como en cantidad de ciclos del mismo.

Así, finalmente, podemos concluir que el mejor algoritmo criptográfico a utilizar, según las métricas analizadas en este proyecto, sería el TEA con 32 bits de tamaño de palabra y 32 rondas.

5 Conclusiones y recomendaciones

Conclusiones

- Fue posible implementar 2 algoritmos criptográficos en un FPGA haciendo uso del lenguaje de descripción de hardware Verilog.
- De los análisis individuales podemos concluir que:
  - El consumo de *slice L*, *slice M*, *slice registers* y *LUTs as logic* crece exponencialmente conforme se varía el tamaño de palabra.
  - En el caso del algoritmo RC5 se puede concluir que la variación del tamaño de llave no afecta el consumo de recursos del FPGA y que la variación en la frecuencia máxima tiene un comportamiento exponencial inverso.
  - En el caso del algoritmo TEA la frecuencia máxima tiene un comportamiento lineal decreciente.
  - Idealmente el algoritmo TEA debe ser implementado para tamaños de palabra menores o iguales a 32 bits, realizando un análisis entre cuanta seguridad y velocidad se desea.
- Del análisis comparativo se puede concluir que de ambos algoritmos, el que consume la menor cantidad de recursos y presenta la mayor frecuencia de reloj es el TEA con 32 bits de tamaño de palabra y 32 rondas. Esto usando como referencia los artículos que describen ambos algoritmos para elegir el punto comparativo del algoritmo DES.

Recomendaciones

- Agregar a las métricas de análisis, el tiempo de procesamiento en ciclos de reloj de cada una de las implementaciones realizadas.
- Mejora el diseño mediante el uso de *pipeline*, esto mejorará el desempeño de los algoritmos y se podría estudiar como este cambio impacta en las métricas analizadas por este trabajo.
- Se prodría profundizar más en el desarrollo y diseño con FPGAs para habilitar el uso de bloques de DSP del mismo y así mejorar el desempeño

y consumo de los algoritmos. Esto para explotar las capacidades de un FPGA.

- Realizar un estudio previo sobre criptografía y más que todo sobre criptoanálisis para tener un mejor criterio sobre como afectan las variaciones del tamaño de palabra, número de rondas y el tamaño de llave en la seguridad del algoritmo.

Bibliografía

Apricorn (2015). Software vs hardware encryption. Recuperado de `http://www.apricorn.com/software-vs-hardware-encryption/`. [Consulta 1 set. 2015].

David J. Wheeler, R. M. N. (1994). *TEA, a Tiny Encryption Algorithm.* Computer Laboratory, Cambridge University.

Denning, D. (1982). *Cryptography and Data Security.* Addison Wesley Publishing Company, Inc.

Egyetem, M. (2015). Field programmable gate arrays (fpga). Recuperado de `http://mazsola.iit.uni-miskolc.hu/cae/docs/pld1.en.html`. [Consulta 12 oct. 2015].

Hamilton, D. (2015). Five data leak nightmares. Recuperado de `http://www.thewhir.com/web-hosting-news/uc-browsers-lack-of-encryption-could-have-leaked-personal-data-to-spies`. [Consulta 16 oct. 2015].

RAE (2015a). Definición criptoanálisis. Recuperado de `http://buscon.rae.es/drae/srv/search?val=criptoan\%E1lisis`. [Consulta 10 oct de 2015.].

RAE (2015b). Definición criptografía. Recuperado de `http://buscon.rae.es/drae/srv/search?val=criptograf\%EDas`. [Consulta 10 oct de 2015.].

Rivest, R. L. (1997). *The RC5 Encryption Algorithm.* MIT Laboratory for Computer Science.

SANS (2007). Hardware versus software a usability comparison of software-based encryption with seagate drivetrust hardware-based encryption. Recuperado de `http://www.seagate.com/staticfiles/SeagateCryptofaceoff.pdf`. [Consulta 1 set. 2015].

Schneier, B. (1996). *Applied Cryptography, Second Edition: Protocols, Algorthms, and Source Code in C.* John Wiley & Sons, Inc.

Vance, J. (2008). Five data leak nightmares. Recuperado de `http://www.networkworld.com/article/2289232/lan-wan/five-data-leak-nightmares.html`. [Consulta 16 oct de 2015.].

Xilinx (2011). Basic fpga architectures. Recuperado de `http://web.mit.edu/clarkds/www/Files/slides2.pdf`. [Consulta 12 oct. 2015].

Xilinx (2015a). 7-series clb architecture overview. Recuperado de `http://www.xilinx.com/training/fpga/7_series_CLB_architecture_video.htm`. [Consulta 12 oct. 2015].

Xilinx (2015b). Field programmable gate array (fpga). Recuperado de `http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm`. [Consulta 12 oct. 2015].

Xilinx (2015c). Fpga vs. asic. Recuperado de `http://www.xilinx.com/fpga/asic.htm`. [Consulta 12 oct. 2015].

Xilinx (2015d). What is a fpga? Recuperado de `http://www.xilinx.com/fpga/`. [Consulta 12 oct. 2015].

Yadron, D. y Beck, M. (2015). Health insurer anthem didn't encrypt data in theft. Recuperado de `http://www.wsj.com/articles/investigators-eye-china-in-anthem-hack-1423167560`. [Consulta 16 oct. 2015].

A Apéndice

Con la finalidad de cuidar el ambiente ahorrando papel, el código realizado se encuentra disponible en `https://github.com/aglt93/proyectoElectrico`.