UNIVERSIDAD D SEVILLA

Escuela Superior de Ingenieros

Departamento de Ingeniería Electrónica

Proyecto Fin de Carrera

REPRODUCCIÓN DE ATAQUES DE FALLOS EN CORES CRIPTOGRÁFICOS BASADOS EN HDL

Ingeniería Superior de Telecomunicaciones

Autor: José Manuel Martín Valencia

Tutor: Hipólito Guzmán Miranda

ESCUELA SUPERIOR DE INGENIEROS

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

UNIVERSIDAD DE SEVILLA

PROYECTO DE FIN DE CARRERA

Reproducción de ataques de fallos en cores criptográficos basados en HDL

Autor: José Manuel Martín Valencia

Tutor: Hipólito Guzmán Miranda

A mi familia por su apoyo
incondicional gracias al cual he podido
afrontar este camino contribuyendo a
ser lo que soy ahora

Resumen

En la actualidad los algoritmos criptográficos son usados como núcleo de muchas aplicaciones computacionales. Esta tecnología aparece en un gran número de aplicaciones cotidianas como en comunicaciones seguras, almacenamiento de datos confidenciales o privados, control de acceso o smart cards usadas en bancos. Para mejorar la velocidad de cálculo y el consumo de potencia, es común que estos algoritmos estén implementados en procesadores hardware específicos para cada aplicación, por ejemplo en smart cards.

Estas implementaciones físicas de los algoritmos criptográficos son sensibles a los denominados ataques de fallos, los cuales pueden ser inducidos de diversas maneras, como por ejemplo por proyección láser sobre el chip de silicio, con el fin de cambiar el estado interno del dispositivo. De esta forma un atacante puede llevar a cabo inyecciones de errores obteniendo así parejas de textos encriptados correcta e incorrectamente. De esta forma, es posible deducir información interna del sistema, en particular los bits de la clave secreta.

El presente proyecto versa sobre la reproducción del ataque de fallos "Differential Fault Analysis" en una implementación hardware del agoritmo "Data Encryption Standard" (DES). Se reproduce el ataque tanto en simulación, para estudiar la viabilidad del mismo, como en un prototipo FPGA (Field Programmable Gate Array). Se concluye el documento aportando diversas ideas sobre cómo proteger estos circuitos a raíz de los resultados obtenidos.

.

Abstract

Nowadays cryptographic algorithms are at the core of many secure computation applications. This technology appears in a number of everyday applications such as secure communications, confidential or private data storage, access control or the smart cards used for banking. To improve computation speed and power consumption, it is common for these algorithms to be implemented into application-specific hardware processors, for example in smart cards.

These physical implementations of cryptographic algorithms are sensitive to the so-called Fault Attacks, which can be induced in various ways, for example, by projecting laser light on the silicon die, in order to change the internal state of the device. Thus an attacker can perform error injections in order to obtain pairs of correctly and faulty ciphertext. This way it is possible to infer internal system information, in particular the bits of the secret key.

This project is about reproduction fault attack "Differential Fault Analysis" in a hardware implementation of the algorithm "Data Encryption Standard" (DES). The attack is reproduced in simulation to study its viability, and on a FPGA (Field Programmable Gate Array). prototype. The project concludes with the contribution of different ideas about how to protect these circuits from the results.

Índice General

1. Introducción

Este proyecto versa sobre el estudio de un determinado algoritmo criptográfico, el Data Encryption Estándar (DES), con el fin de reproducir ataques de fallos sobre dicho algoritmo. Estos fallos darán la posibilidad a un atacante para poder vulnerar el sistema y extraer información secreta. A grandes rasgos podría parecer que lo que se pretende es romper la seguridad de un sistema per se. Esto no es así ni muchísimo menos, pues aunque se busque vulnerar la seguridad de un sistema, cosa propia de los atacantes maliciosos, se hará con el objetivo de reproducir los ataques a nivel de diseño e implementación digital, de forma que un ingeniero pueda estudiar cómo proteger el circuito frente a dichos ataques.

1.1 Revisión histórica

La historia de la criptografía [1] se remonta desde las primeras guerras donde la información que se transportaba debía ser secreta. El primer sistema de criptografía data del siglo V a.C conocido como "Escítala. El segundo llamado "Cifrado César" estaba basado en una tabla de sustitución y fue usado por los romanos.

En la historia destacan importantes criptógrafos como Leon Battista Alberti en 1465 que inventó un sistema de sustitución polialfabética y Blaise de Vignère que escribió un importante tratado sobre "la escritura secreta". En los siglos XVII, XVIII y XIX, los monarcas se interesaban más por la criptografía ya que información de vital importancia ofrecía una importante ventaja estratégica. Por ejemplo el ejército de Felipe II empleó un sistema de criptografía basado en un alfabeto de más de 500 símbolos que fue roto por los matemáticos del rey de Francia, Enrique IV, ofreciéndole a éste la capacidad de vencer los ejércitos del rey de España. Esto dio lugar a una queja por parte de España ante el papa Pio V de usar magia negra.

A partir del siglo XX la criptografía se volvió primordial en los dos grandes acontecimientos que marcaron el siglo: la Primera y Segunda Guerra Mundial. Durante estas guerras se dio un gran avance en la criptografía, pues surgieron las máquinas de cálculo que facilitaban el trabajo de encriptar y desencriptar. De entre esas máquinas destacó la máquina Enigma usada por los alemanes durante la Segunda Guerra Mundial que automatizaba los cálculos y supuso un esfuerzo por parte de los matemáticos del bando aliado para poder romper su sistema de cifrado. Es a partir de ese momento

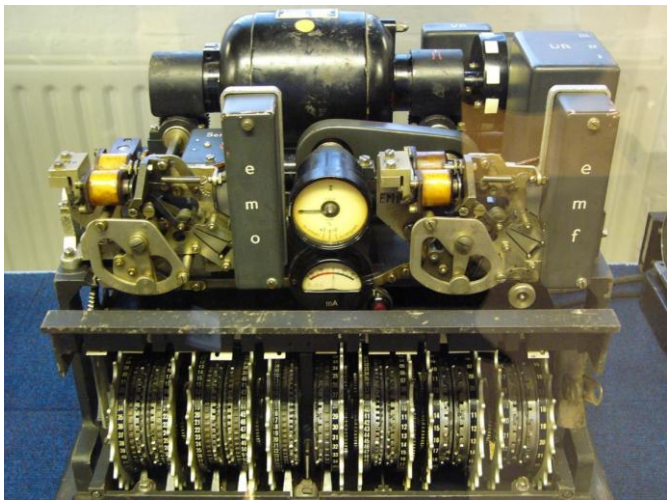


Figura 1. La máquina alemana de cifrado Lorenz, usada en la Segunda Guerra Mundial para el cifrado de los mensajes para los generales de muy alto rango. Imagen liberada al dominio público por el usuario "Matt Crypto" de la wikipedia inglesa.

que la criptografía empieza a desarrollarse desde un punto de vista teórico importante.

Las bases adoptadas para la criptografía tras la Segunda Guerra Mundial fueron postuladas por Claude Shannon ya que sus investigaciones sobre la teoría de la información fueron hitos esenciales que permitieron grandes avances teóricos. Además, los avances en la computación automática suponen un importante desarrollo de nuevos sistemas. A mediados de los 70, la Oficina Nacional de Normalización norteamericana (NIST: National Institute of Standards and Technology) publicó el primer diseño lógico de un cifrador que luego se convertiría en el principal sistema criptográfico hasta finales de siglo: El Encriptador de Datos Estándar (DES). En esas fechas ya se empezaba a iniciar la última revolución de la criptografía teórica y práctica: los sistemas asimétricos, resultando un salto cualitativo importante que permitió introducir la criptografía en otras áreas esenciales, como en la firma digital.

Hoy día con la aparición de la informática y el fuerte uso de las comunicaciones digitales se ha detectado un progresivo problema de seguridad, pues las transacciones que se realizan a través de la red pueden ser interceptadas y por lo tanto se puede comprometer la información que debe ser garantizada. Es por ello que los principales objetivos de la criptografía se han visto superados y por ello la criptografía ha pasado a formar parte de la criptología que se encargará de los algoritmos, protocolos y sistemas usados en conjunto para proteger la información dotando así de seguridad a las comunicaciones.

1.2 Seguridad y sistemas seguros (smartcards, llaves electrónicas, …)

El término seguridad se refiere generalmente a la desaparición del riesgo y por lo tanto a la confianza en algo o alguien, pudiendo tomar diversos sentidos según el área al que haga referencia. En lo que respecta al proyecto la seguridad se centrará al ámbito informático y más concretamente a los sistemas de seguridad, basados en encriptaciones hardware implementadas sobre el propio circuito. Un ejemplo de estos sistemas seguros son las smartcards, las cuales son tarjetas con circuitos integrados que permiten la ejecución de cierta lógica programada. En este caso de entre las distintas smartcards que existen, según el ámbito del proyecto interesa conocer aquellas que implementen alguna lógica de seguridad. Estas smartcards contienen unos módulos hardware necesarios para la ejecución de algoritmos usados en sistemas de encriptación y firmas digitales, pudiendo almacenar de forma segura un certificado digital y siendo capaz de firmar documentos o autentificarse. Todas estas operaciones se llevarán a cabo gracias a una clave privada almacenada en la propia tarjeta convirtiendo a esta clave en el objetivo a abordar [2].

Un ejemplo de smartcard de este tipo son las tarjetas SIM o GSM, compuestas de un procesador criptográfico seguro, sistema de archivos seguro, etc; que proveen de servicios de seguridad.

Otro ejemplo de sistema de seguridad son las llaves electrónicas usadas como medio de acceso a cualquier propiedad, como puede ser en un vehículo.

En general estos sistemas de seguridad deben implementar el hardware apropiado para que el cifrado sea lo más seguro posible empleando para ello una clave secreta.

De esta forma uno puede entender el nivel de importancia que puede tener la seguridad en sistemas críticos sobre los que se apoya gran parte de la civilización moderna y así se justifica la razón por la cual se aborda este estudio en el proyecto.

1.3 Objetivo del proyecto

El proyecto se basa fundamentalmente en reproducir un ataque de fallos en cores criptográficos seguros tales como el Data Encryption Standard (DES) y así simular un intento de ruptura de seguridad, esto es, conocer la clave secreta, con el fin de poder proponer contramedidas frente a diversos tipos de ataques. Para ello será necesario conocer en detalle el algoritmo a explotar con el fin de averiguar las vulnerabilidades frente a diversos tipos de ataques. De este modo se elegirá un tipo de ataque sobre el que se fundamentará el estudio y la puesta en práctica.

1.4 Organización de la memoria

La memoria de este proyecto está distribuida en 10 capítulos que abordarán diversos aspectos a tratar para una correcta comprensión del tema a analizar.

El primer capítulo muestra una introducción del tema central del proyecto desde el punto de vista histórico de cómo la criptografía ha influido a través de los tiempos hasta la actualidad, donde prácticamente en cualquier transacción se emplean sistemas seguros. También se presentan los objetivos principales de la criptografía y finalmente el objetivo que se pretende alcanzar.

El segundo capítulo define el concepto de criptografía, presentando los tipos principales y los objetivos sobre los que se fundamentan estos sistemas.

El tercer capítulo explica los diferentes tipos de ataques en un sistema de seguridad, haciendo hincapié en el tipo de ataque que se desarrolla en el proyecto.

El cuarto capítulo describe en detalle el algoritmo a analizar.

El quinto capítulo aborda de forma teórica el ataque elegido sobre el algoritmo criptográfico escogido, presentando varias formas de llevarlo a cabo, para finalmente optar por uno y así llevarlo a la práctica.

El sexto capítulo pone en práctica el ataque de fallos elegido sobre el core criptográfico. Para ello, se instrumentalizará el código en VHDL[1] para realizar ataques de fallos de manera automática comprobando así la viabilidad de dicho ataque y obtener resultados cuantitativos tales como el número de ataques, los diferentes tipos de errores, etc

[1] VHDL es el acrónimo de la combinación de VHSIC (Very High Speed Integrated Cicruit) y HDL (Hardware Description Language).

El séptimo capítulo lleva a cabo el ataque de fallos sobre el algoritmo criptográfico implementado en una FPGA mediante la herramienta FT-UNSHADES2[2], para luego hacer un post-procesado en python con los datos obtenidos en dicha herramienta.

El octavo capítulo presenta las conclusiones y posibles trabajos futuros como la aplicación a algoritmos más sofisticados como el Advanced Encryption Standard (AES).

El noveno capítulo recoge las referencias bibliográficas sobre las que se basa el proyecto.

Finalmente, se incluyen como anexos los pasos necesarios para llevar a cabo el ataque en simulación y en FPGA.

[2] Acrónimo de Fault Tolerance University of Sevilla Hardware Debugging System.

2. Conceptos

El proyecto trata sobre criptografía, abordando ciertos temas cuyo conocimiento debe quedar claro. Ésta es la razón por la cual se presentan de forma resumida ciertos conceptos generales sobre criptografía.

2.1 Definición de criptografía

La criptografía se define como el estudio encargado de cambiar las representaciones lingüísticas de ciertos mensajes con el fin de ocultarlos haciéndolos incoherentes a receptores no autorizados, por lo que la confidencialidad de los mensajes es primordial. Para ello, se han diseñado sistemas de cifrado cuyo propósito es la encriptación. Para ello se necesitará de un mensaje a encriptar, también conocido como texto plano y una clave usada por las operaciones pertinentes con el fin de transformar el texto plano de entrada en otro texto completamente distinto. También comprende otros conceptos como no-repudio, autentificación, confidencialidad e integridad, que están fuera del alcance del presente trabajo.

Tipos de criptografía

En el ámbito de la criptografía existen diferentes tipos de algoritmos criptográficos. De entre ellos se destacarán dos principalmente:

Criptografía simétrica o criptografía de clave secreta [3]

La criptografía simétrica describe un método criptográfico basado en el uso de una sola clave tanto para encriptar como para desencriptar, cambiando para ello el orden de las operaciones.

Algunos ejemplos de estos sistemas son: DES [4], 3DES [5], RC5 [6], AES[7], Blowfish [8] e IDEA [9].

Según la cantidad de bits del mensaje a encriptar, existen dos tipos de cifradores:

- **Cifradores de flujo** [10]: cifran el mensaje bit a bit, en situaciones tales como conversaciones telefónicas donde los fragmentos de datos a encriptar en tiempo real pueden llegar a ser muy pequeñas como 8 bits o hasta 1 bit. Un ejemplo de este cifrado es el RC4[11].

- **Cifradores de bloque** [12]: cifran el mensaje dividiéndolo en bloques de n bits, dando lugar a dos situaciones, dependiendo de si el tamaño del mensaje es mayor o menor de n (tamaño predefinido a cifrar):

- Si es mayor, se usarán los modos de operación que son métodos que se encargan de encriptar cadenas de mensajes mayores que n, dividiendo el mensaje en sucesivos bloques a encriptar de tamaño n. De esta forma cada bloque de tamaño n se encripta de forma separada del resto, siguiendo diferentes esquemas de cifrado que gestionan dichos bloques, tales como: Electronic Code-Book (ECB), Cipher feedback (CBC), Propagating Cipher-Block Chanining (PCBC), Output FeedBack (OFB).

  Además se podrá aplicar un esquema de relleno para el caso de que el bloque a encriptar no sea múltiplo entero del tamaño de bloque.

- Si es menor, sólo se aplicará el esquema de relleno.

Ejemplos de este tipo de cifrado son: DES y AES.

La seguridad de este tipo de criptografía se basa en el secretismo de la clave y no del algoritmo tal como expone la Máxima de Shannon ya que los algoritmos son públicos o de código abierto por lo que cualquiera puede tener acceso a ellos y poder estudiarlos con el fin de encontrar alguna debilidad estructural. Así que la última barrera entre el atacante y la clave se basa en el secretismo de dicha clave, lo que plantea un problema en cuanto a la distribución de las claves puesto que se puede comprometer la seguridad del sistema ya que se corre el riesgo de poder descubrir la clave en el canal de comunicación donde se transmite la misma. Por lo tanto ante este inconveniente surgen la criptografía asimétrica y la híbrida, que aunque son más lentas, son más seguras.

Criptografía asimétrica o criptografía de clave pública [13]

La criptografía asimétrica surgió ante el inconveniente del intercambio de claves en sistemas criptográficos simétricos y se basa en el uso de dos claves, una privada que sólo conoce el receptor y una pública generada por el propio receptor a partir de la privada y que conoce todo el mundo ya que el mismo receptor la puede distribuir libremente.

En este sistema criptográfico existen dos modos de operación:

- Encriptacion con clave pública: un mensaje encriptado con la clave pública sólo puede ser desencriptado usando una clave privada (a partir de la cual se generó la clave pública), asegurando la confidencialidad.
- Firma digital: un mensaje encriptado con la clave privada puede ser desencriptado por cualquiera que conozca la clave pública, asegurando la autoría del mensaje.

Sin embargo el principal inconveniente que presenta este sistema es un mayor tiempo de procesado en comparación con los sistemas simétricos, ya que estos algoritmos son bastante más complejos, las claves son de mayor tamaño y el mensaje resultante es mayor que el original.

Ejemplos de estos sistemas son: Diffie-Hellman [14], RSA [15], DSA [16], ElGamal [17], Criptografía de curva elíptica [18] y Criptosistema de Merkle-Hellman [19].

2.2 Objetivos de la criptografía

La criptografía debe dotar de seguridad a los sistemas proporcionando una serie de propiedades:

Confidencialidad: la información está accesible únicamente a personal autorizado, empleando códigos y técnicas de cifrado.

Integridad: la información no debe ser alterada y reproducirse con exactitud.

Vinculación: la información está vinculada a un emisor ya sea una persona o un sistema, empleando para ello por ejemplo la firma digital.

Autenticación: el acceso a la información se hace a través de mecanismos que permitan verificar la identidad del comunicador.

Sin embargo, no siempre son necesarias todas estas propiedades ya que dependienderá de la aplicación [20].

3. Tipos y revisión de ataques a cores criptográficos

Los algoritmos criptográficos no son perfectos al 100%, pues se puede extraer información del propio algoritmo mediante diversas técnicas, tales como un análisis del consumo de potencia o un fallo inyectado en una determinada zona del circuito que altere el normal funcionamiento del mismo. Estas técicas permiten inducir información acerca de la clave.

A continuación aparecen las diferentes técnicas que ayudan a extraer información relevante para la obtención de la clave.

3.1 Side Channel Analysis

Este ataque se basa en información obtenida de la implementación del dispositivo criptográfico. Dicha información es obtenida a partir de un canal de información adicional al propósito principal del algoritmo como por ejemplo: información de temporización, el consumo de potencia, emanaciones electromagnéticas o el sonido. A esta información no intencionada que escapa del circuito se le suele llamar fuga o "leakage" y por esta razón este tipo de análisis se denomina "Análisis de Canal Lateral".

3.1.1 Simple Power Analysis

Simple Power Analysis (SPA) [21] es un ataque de canal lateral basado en el análisis del consumo de potencia del dispositivo donde el atacante puede deducir el estado interno del dispositivo observando información extraída de dicho consumo durante el normal funcionamiento del dispositivo. La primera publicación que menciona dichos ataques es [22].

En este apartado sólo se pretende dar a conocer este ataque como paso preliminar para un ataque directo sobre el dispositivo, ya que este tipo de ataque permite conocer las operaciones llevadas a cabo en función de la potencia consumida por el propio circuito. Sin embargo el principal inconveniente de este ataque es la limitada resolución para identificar determinadas operaciones específicas tales como una determinada ronda de encriptación ya que en muchas ocasiones se identifica a las rondas por descarte en lugar de identificarlas por las propias operaciones que las componen.

3.1.2 Differential Power Analysis

Differential Power Analysis (DPA) [21] es una extensión del ataque anterior en el cual el atacante lleva a cabo una intensiva adquisición de datos con el fin de analizar estadísticamente las trazas de consumo de potencia para descubrir la clave. Este proceso empieza mediante la selección de las subclaves que el atacante desea recuperar, donde cada una representa una pequeña parte de la clave total. De este modo el proceso consiste en tres pasos:

1. Se usan diferentes textos planos y subclaves candidatas, para predecir valores intermedios dentro de la implementación objetivo, en función de los valores que toman dichos textos planos y subclaves.
2. Para cada uno de los valores anteriores, el atacante modela el consumo energético. Este modelado dependerá de la tecnología en la que esté implementado el algoritmo.
3. Para cada una de las subclaves estudiadas, se compara el modelo de fugas obtenido anteriormente con el valor medido al alimentar el sistema real con los mismos textos planos. Dado que en el consumo de potencia del circuito influyen más factores que la operación que se está modelando, se usa un distinguidor estadístico como puede ser el correlador de Pearson, el cual muestra el grado de relación entre las subclaves candidatas y la subclave original (parte de la clave total).

3.2 Fault Analysis

El análisis de fallos es una técnica usada en los cifradores de bloques, basado en producir un error computacional que altere el resultado en la salida, es decir, una corrupción de los elementos de memoria internos. En una implementación hardware, esto significa en un cambio de los valores de los registros (flip-flops) del circuito que serán atacados, durante el normal funcionamiento del mismo. Dichos ataques pueden ser inducidos mediante una proyección láser sobre el chip de silicio [23], aunque también existe la posibilidad de usar radiación de iones [24], el cual es un método para atacante con suficiente presupuesto.

En los siguientes apartados se mostrarán varios tipos de análisis de fallos.

3.2.1 Differential Fault Analysis

Differential Fault Analysis (DFA) [25] es una técnica que consiste en atacar el sistema (en un determinado punto) provocando un funcionamiento anómalo del circuito, de forma que el error inyectado se propague a la salida, produciendo una salida errónea (C^{faulty})[3]. Dicha salida obtenida a partir de un texto plano se compara con la salida del mismo texto plano encriptado correctamente (C^{gold})[4]. De este modo y dado que el circuito ha funcionado correctamente en los dos casos antes del fallo, el atacante podrá inferir información sobre las últimas transformaciones, por ejemplo sobre los bits afectados por el error inyectado y la salida de la última ronda de transformación.

Las aplicaciones de DFA han demostrado que algoritmos como el Data Encryption Estándar (DES) [26] y Advanced Encryption Estándar (AES) [27,28] son vulnerables frente a este tipo de ataques por lo que ése será el objeto de estudio de este proyecto, conocer cómo afectan los ataques de fallos a las implementaciones hardware de algoritmos criptográficos seguros.

[3] Texto cifrado fallido a causa del error inyectado.
[4] Texto cifrado de forma correcta en ausencia de errores.

El algoritmo DES ha resultado ser inseguro frente a ataques de fuerza bruta, donde se comprueba la clave de manera secuencial, por lo que parecería absurdo centrarse en estudiar este algoritmo pues ya apenas se usa. Sin embargo el estudio de este proyecto, primero a nivel matemático para más adelante estudiarlo a nivel de implementación hardware, se basa en entender cómo la propagación de un error del circuito puede ser aprovechada para vulnerar la seguridad y extraer información del propio sistema. Así pues se estudiará el algoritmo DES frente a este tipo de ataque por ser el sistema de cifrado que precedió al AES (actual esquema de cifrado usado en todo el mundo), lo que proporcionará una visión cualitativa de cómo se comporta un sistema de seguridad ante este tipo de ataques.

3.2.2 Collision Fault Analysis

Collision Fault Analysis (CFA) [25] es un tipo de ataque de fallos que produce una salida errónea, pero a diferencia del anterior este método obtiene información a partir de un texto plano que produce la misma salida (encriptada correctamente) comparada con la errónea. De este modo, se produce una colisión en la salida con dos textos idénticos a partir de dos textos planos distintos cada uno encriptado de forma correcta e incorrecta. Este hecho puede parecer difícil de producirse pues el algoritmo criptográfico suele estar diseñado para que la salida sea lo más aleatoria posible. Sin embargo dicha dificultad es mitigada si el fallo se produce al principio del algoritmo, ya que es posible evitar que el efecto de avalancha de las encriptaciones de ambos textos planos altere significativamente los textos orígenes. De esta forma un fallo al principio del algoritmo provoca un efecto más predecible lo que permite inferir información de las primeras operaciones llevadas a cabo sobre el texto origen.

3.2.3 Ineffective Fault Analysis

Ineffective Fault Analysis (IFA) [25] es un método ligeramente diferente al anterior que trata de encontrar un par de textos planos de tal forma que al encriptarlos de forma correcta e incorrecta (con un error en una operación precisa) coincidan sus salidas. De este modo dicho método gana información a partir de los fallos que no modifican localmente los valores intermedios a donde van dirigidos los datos con fallo. Esto hace que las salidas sean idénticas y de ahí origina su nombre dicho método.

Como se puede observar se asemeja a CFA en cuanto a que las salidas deben ser idénticas, sin embargo difiere en que CFA obtiene información a partir de un solo fallo buscando para ello textos planos que permitan la coincidencia final. Usando IFA se necesita comparar pares de textos planos usando tantos fallos como sean necesarios hasta producir un fallo sin efecto en la salida.

3.2.4 Safe-Error Analysis

Safe-Error Analysis (SEA) [25] es el último método de análisis de fallos que se basa a diferencia del anterior en modificar el estado interno de determinados bits que no modifican el resultado intermedio de las siguientes operaciones. Esto es posible debido que dichos bits no son usados.

Este método se aplicó por primera vez para romper los criptosistemas RSA [29, 30, 31].

4. Data Encryption Standard (DES)

En este apartado se pretende dar a conocer el algoritmo criptográfico DES junto con las bases matemáticas sobre las que se sustenta dicho algoritmo, con el fin de comprender la vulnerabilidad de este sistema.

4.1 Revisión histórica

El Encriptador de Datos Estándar (DES) [32] es el nombre por el cual se conoce al cifrado de bloques que se pretende analizar. Dicho algoritmo fue diseñado en los años 70, cuando la Oficina Nacional de Normalización de Estados Unidos en Mayo de 1973 buscaba un algoritmo criptográfico que pudiera asegurar las comunicaciones entre compañías. Sin embargo ningún candidato fue aceptado hasta que en Agosto de 1974 IBM presentó un cifrado de bloques diseñado por el criptógrafo Horst Feistel, llamado Lucifer. Dicho algoritmo está basado en una red de sustitución, método muy empleado en cifrados en bloque, con una estructura particular que permite encriptar y desencriptar empleando las mismas operaciones en el mismo orden, invirtiendo únicamente el orden de subclaves empleadas. Esto permitió que su implementación fuera sencilla en hardware y así fue como Lucifer, tras una serie de modificaciones por parte de la Agencia de Seguridad Nacional (NSA), se convirtió en un estándar oficial en 1976. De esta forma Lucifer se convirtió en el algoritmo DES permitiendo asegurar las comunicaciones gubernamentales. Sin embargo pronto se comprobó que la corta longitud de la clave podía ser motivo de una brecha de seguridad mediante una búsqueda intensiva de claves. Por esta razón fue sustituido por Triple-DES en 1998, el cual consiste en tres sucesivas rondas de encriptación empleando para ello una clave total tres veces más larga. Hoy en día Triple-DES sigue usándose en los protocolos de seguridad de las smart card EMV (Europay MasterCard Visa) aunque está desapareciendo lentamente, siendo reemplazado por el nuevo Encriptador Avanzado Estándar (AES), adoptado en 2002.

4.2 Esquema de funcionamiento

DES es un algoritmo de cifrado en bloque que usa un texto plano de 64 bits para transformarlo, mediante operaciones de reemplazo y sustitución, en un texto completamente diferente de 64 bits, empleando una clave de 64 bits. Dicha clave aunque mide 64 bits en realidad sólo se emplea 56 bits, dado que los 8 bits restantes son usados como bits de paridad y posteriormente son descartados, y por lo tanto no son usados para encriptar. La figura 2 muestra el esquema completo.

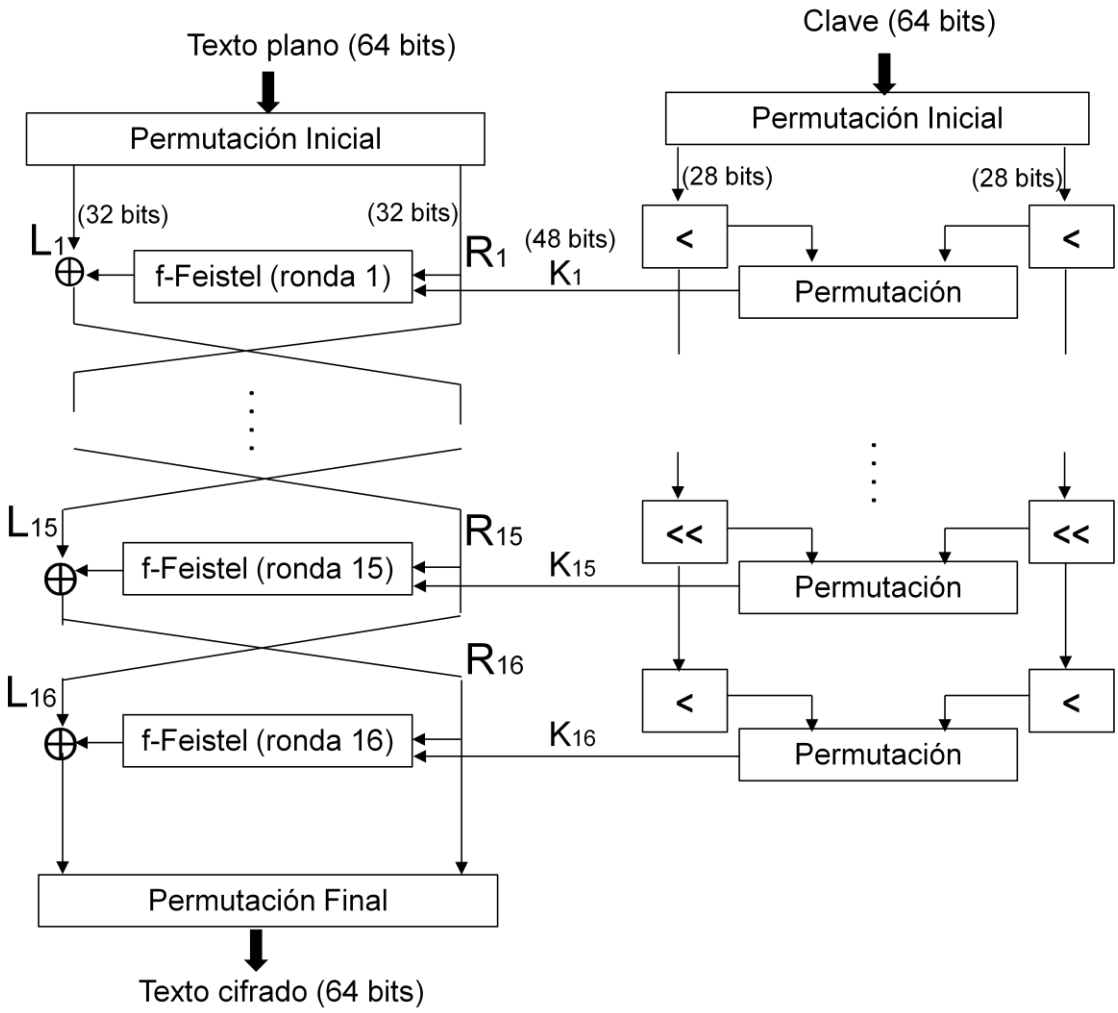


Figura 2. Esquema completo del algoritmo DES.

4.2.1 Estructura básica

El algoritmo se basa en la sucesión de 16 rondas de cifrado consecutivas, repitiendo el mismo esquema básico de cifrado además de una permutación inicial y final del proceso completo, como ilustra la figura 2. Dicha estructura básica se basa en dividir el bloque de 64 bits en sus dos mitades izquierda y derecha, de tal forma que la mitad derecha de la ronda i-ésima (R_i) se transformará en otro texto de 32 bits distinto, empleando una red de sustitución, llamada red de Feistel en cada ronda. Estos 32 bits de salida de la red de Feistel se combinan con los otros 32 bits de la mitad izquierda de la ronda actual (L_i) mediante una operación XOR dando lugar una nueva mitad izquierda conservando la mitad derecha sin modificaciones. A continuación ambas mitades se intercambian y pasan como entradas a la siguiente ronda donde se repite el mismo proceso.

4.2.2 Estructura completa

El proceso completo de encriptación incluye:

1. Permutación inicial del texto plano: al principio del algoritmo se lleva a cabo una permutación sobre los 56 bits de la clave una vez se han descartado los 8 bits de paridad. Dicha permutación altera el orden de los bits tal como aparece a continuación en la figura 3 :

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Permutación inicial del texto plano

57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7
56	48	40	32	24	16	8	0
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6

Figura 3. Permutación inicial del algoritmo DES sobre el texto plano.

La figura 3 muestra el resultado de permutar inicialmente un texto plano, viéndolo como una matriz de 8x8 bits, donde cada columna se transforma una fila tras la permutación.

2. 16 rondas de cifrado: se repiten las 16 rondas consecutivamente, encriptando de manera sucesiva las salidas de cada ronda en la siguiente:

$$\bigoplus_{i=1}^{16} F(k_i, R_i) \oplus L_i \qquad (1)$$

Donde:

- $F(k_i, R_i)$: función de Feistel de la ronda i-ésima.
- k_i : clave de la ronda i-ésima.
- R_i : mitad derecha del texto de entrada en la ronda i-ésima.
- L_i : mitad izquierda del texto de entrada en la ronda i-ésima ($=R_{i-1}$).

3. Permutación final: a la salida de la última ronda se le aplica una permutación final.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Permutación Final

39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25
32	0	40	8	48	16	56	24

Figura 4. Permutación final del algoritmo DES.

4.2.3 Función de Feistel

La función de Feistel, el cual opera en todas las rondas se basa en la ejecución de 4 pasos:

1. **Permutación expansiva**: los 32 bits de entrada se expanden duplicando algunos bits.
2. **Mezcla con la subclave**: los 48 bits obtenidos de la expansión se mezclan con la subclave propia de la ronda, realizándolo por medio de una operación XOR bit a bit.
3. **Sustitución**: el resultado anterior se divide en 8 bloques de 6 bits para poder ser transformadas en 6 bits según la sustitución propia de cada caja de sustitución. La caja de sustitución se encarga de reemplazar los 6 bits de entrada por 4 bits a la salida.
4. **Permutación:** finalmente se combinan todas las salidas de cada caja de sustitución mediante una última permutación.

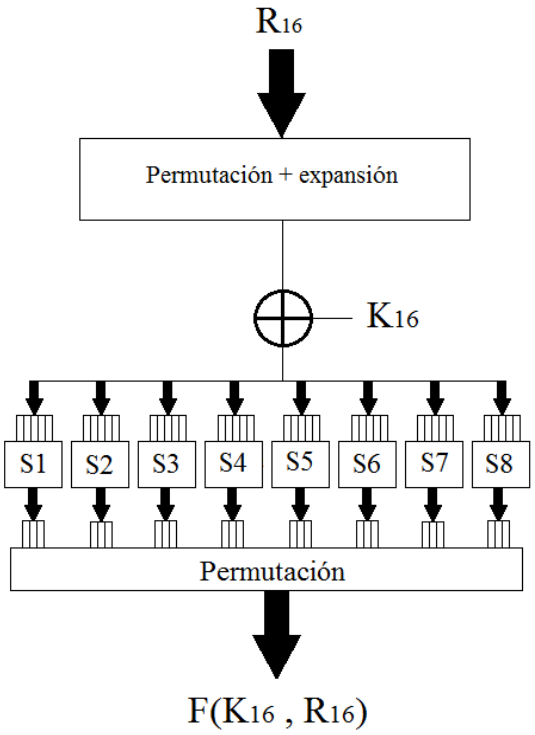


Figura 5. Esquema general de la función de Feistel.

La figura 5.1 muestra de forma más detallada la permutación expansiva y la permutación final de cada ronda.

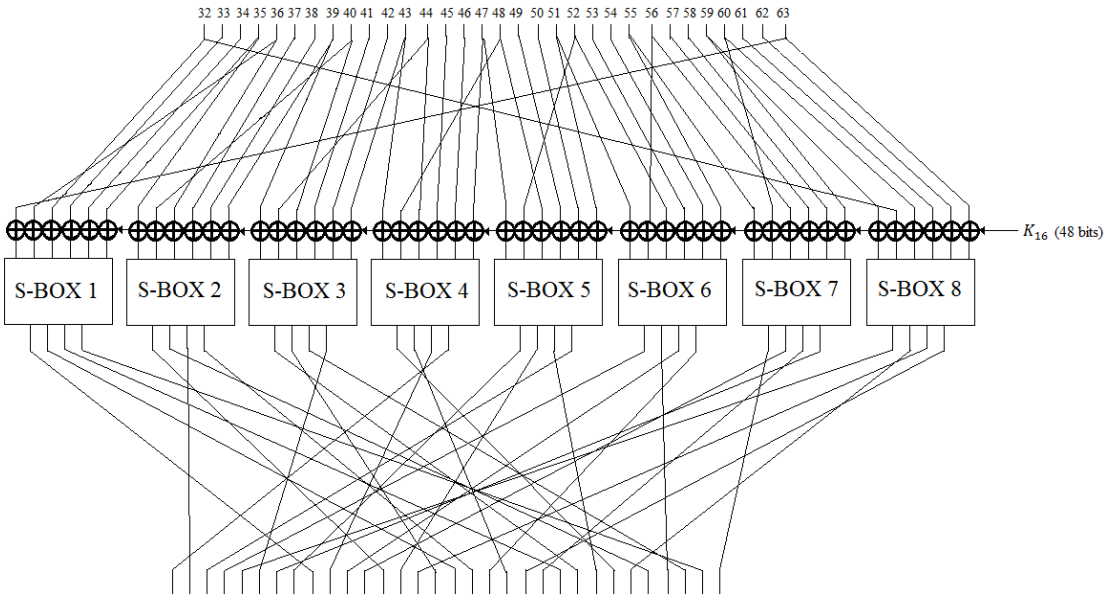


Figura 5.1. Esquema detallado de la función de Feistel.

A continuación viene reflejada matemáticamente la función de Feistel de la ronda i-ésima para la caja de sustitución j-ésima:

$$F(k_i, R_i) = P(S_j(PE_j(R_i) \oplus K_{i,j})). \qquad (2) \qquad \text{, para } j = 1,2,3,4,5,6,7,8$$

Donde:

- $F(k_i, R_i)$: función de Feistel de la ronda i-ésima.
- k_i : clave de la ronda i-ésima.
- R_i : mitad derecha del texto de entrada en la ronda i-ésima.
- L_i : mitad izquierda del texto de entrada en la ronda i-ésima.
- $P(\cdot)$: permutación a la salida de las cajas de sustitución.
- $S_j(\cdot)$: transformación de la caja de sustitución j-ésima.
- $PE_j(\cdot)$: permutación expansiva, que convierte los 32 bits de entrada en 48 bits que entran a las cajas de susitución.
- $K_{i,j}$: subclave de la caja de sustitución j-ésima de la ronda i-ésima.

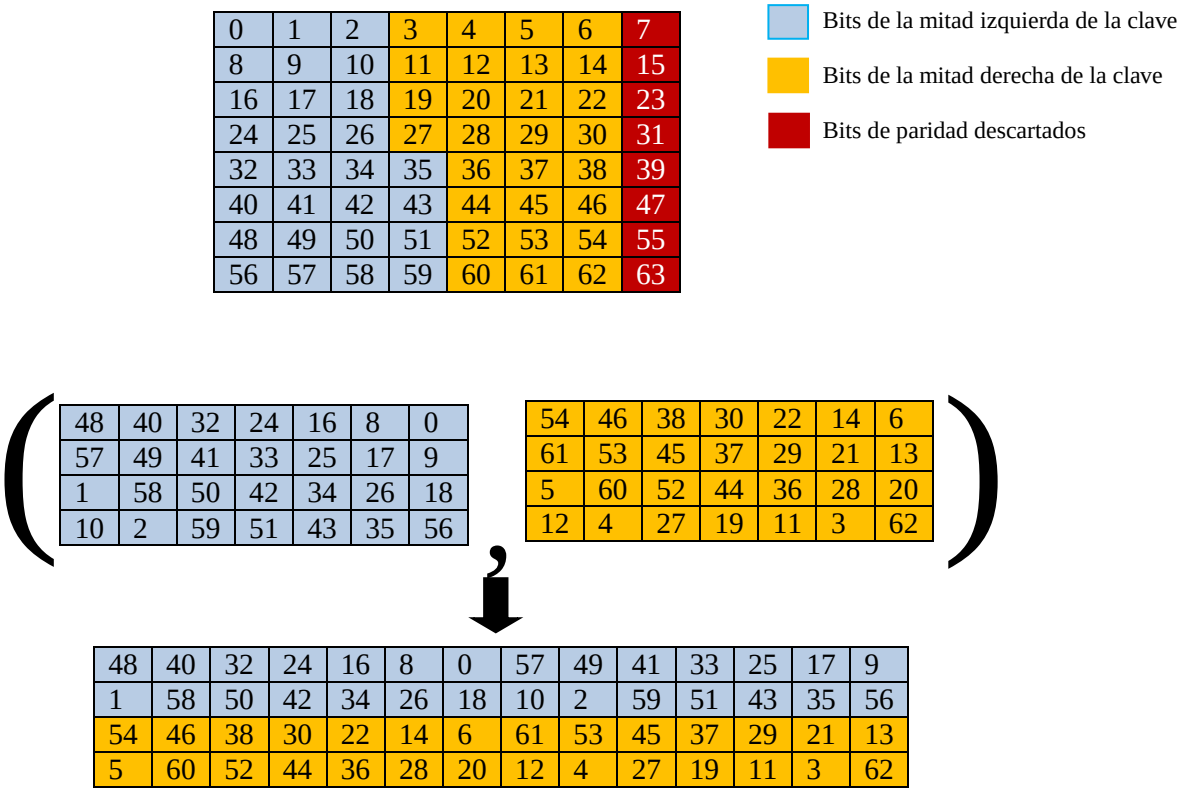
4.2.4 Generación de subclaves

Como se comentó anteriormente el sistema emplea una clave inicial de 64 bits de la cual sólo se usarán 56 bits pues 8 bits son de comprobación de paridad. De modo que a partir de estos 56 bits de clave deben generarse las claves necesarias para cada ronda.

El proceso de generación de la clave se basa en desplazamientos de bits a la izquierda o derecha, dependiendo si el usuario quiere encriptar o desencriptar, respectivamente. Para ello en primer lugar en la primera ronda se aplica a la clave una elección permutada donde la clave se divide en dos mitades de 28 bits asignando bits en función de la permutación.

A continuación en la figura 6 se puede comprobar dicha elección permutada para el caso de que el usuario quiera encriptar.

Permutación inicial de la clave



0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(| 48 | 40 | 32 | 24 | 16 | 8 | 0 |
|---|---|---|---|---|---|---|
| 57 | 49 | 41 | 33 | 25 | 17 | 9 |
| 1 | 58 | 50 | 42 | 34 | 26 | 18 |
| 10 | 2 | 59 | 51 | 43 | 35 | 56 |

54	46	38	30	22	14	6
61	53	45	37	29	21	13
5	60	52	44	36	28	20
12	4	27	19	11	3	62

)

↓

48	40	32	24	16	8	0	57	49	41	33	25	17	9
1	58	50	42	34	26	18	10	2	59	51	43	35	56
54	46	38	30	22	14	6	61	53	45	37	29	21	13
5	60	52	44	36	28	20	12	4	27	19	11	3	62

Figura 6. Permutación de la clave original. El color azul y amarillo constituye la mitad izquierda y derecha, respectivamente, que luego se concatenan. El color rojo simboliza los bits de paridad. La coma representa la concatenación.

Esta clave generada de 56 bits sólo se usa para la primera ronda, pues para dicha ronda no se requiere de ninguna transformación. Para las demás rondas, la clave se rotará una o dos posiciones respecto a la clave de la ronda anterior. La tabla 1 refleja las rotaciones de cada ronda de encriptación.

Ronda	**Nº bits desplazados a la izquierda**
1	0
3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15	2
2, 9, 16, 1	1

Tabla 1. Rotaciones de la clave para cada ronda.

La figura 5 muestra cómo quedaría la clave de la primera ronda tras la elección permutada del inicio del algoritmo. Sin embargo a dicha clave se le deben desechar 8 bits, pues cada caja de sustitución necesitará de 6 bits de clave. Los bits que se descartan serán siempre aquellos que ocupen las posiciones: 8, 17, 21, 24, 34, 37, 42, 53 de la clave de cada ronda.

5. Differential Fault Analysis sobre DES

Este apartado se centra en aplicar un Differential Fault Analysis (DFA) sobre el algoritmo criptográfico DES con el fin de obtener información que permita descubrir la clave. Para ello, en primer lugar se explicará el fundamento matemático detrás de este ataque, para posteriormente llevarlo a cabo mediante simulación en el siguiente capítulo.

5.1 Ataque básico

El ataque consiste en modificar una serie de bits internos de un determinado registro consiguiendo que el error se propague a través de los sucesivos registros hasta modificar el valor de la salida encriptada. De este modo a partir de la salida encriptada correcta y erróneamente se puede inferir información sobre la clave usada en el algoritmo. Ésa es la idea básica sobre la que se basará el planteamiento del proyecto.

5.1.1 Ataque en la ronda 16

El ataque en la ronda 16 [32] consiste en la forma más eficaz de inyectar un fallo para poder ver sus efectos en la salida. El ataque debe ser inyectado justo antes de que la entrada de la ronda 16 se expanda para combinarla con la clave de dicha ronda. Los efectos del error propagado se pueden observar en la figura 7.

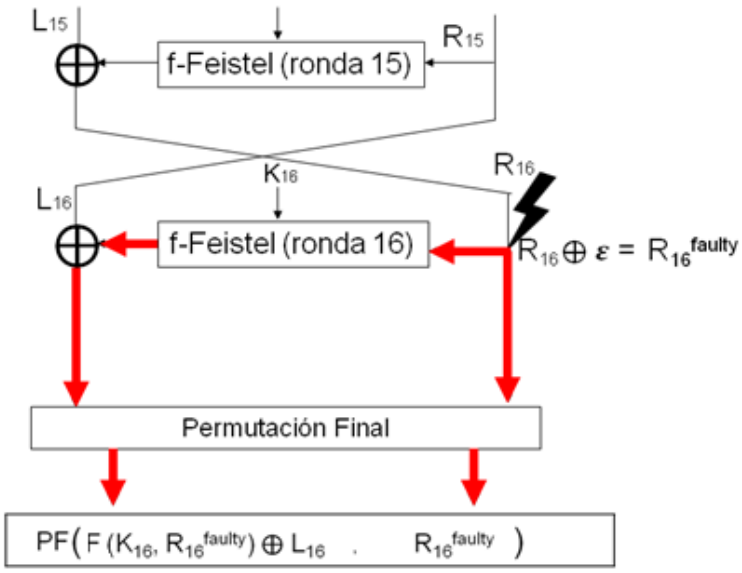


Figura 7. Efecto del error propagado en la última ronda

Matemáticamente el ataque considera que tanto para la encriptación correcta como para la errónea, el texto justo antes del fallo inyectado es el mismo. De este modo el atacante puede deshacer las últimas permutaciones y saber la entrada de la última ronda tanto para el caso correcto como para el erróneo (con el fallo inyectado). De este modo, sólo habría que emular la última ronda para dichas entradas, probando con diversas subclaves para cada caja de sustitución hasta que se verifique la ecuación 4.

$$C^{gold}:\ PF(F(k_{16}, R_{16}) \oplus L_{16}, R_{16}) \quad (3.1)$$

$$C^{faulty}: PF(\ F(k_{16}, R_{16} \oplus \text{error}) \oplus L_{16},\ R_{16} \oplus \text{error}) \quad (3.2)$$

$$PI(C^{gold} \oplus C^{faulty}) = F(k_{16}, R_{16}) \oplus F\left(k_{16}, R_{16}^{faulty}\right), \text{error} =$$

$$= P(S_j(PE_j(R_{16}) \oplus K_{16,j})) \oplus P(S_j(PE_j(R_{16}^{faulty}) \oplus K_{16,j})),\ \text{error} \quad \text{para } j = 1,2,..8 \quad (3.3)$$

Donde:

- PF : permutación final después de la última ronda de encriptación.
- PI : permutación inversa, que anula la permutación final del algoritmo.
- $F(k_i, R_i)$: función de Feistel de la ronda i-ésima.
- k_i : clave de la ronda i-ésima.
- R_i : mitad derecha del texto de entrada en la ronda i-ésima.
- L_i : mitad izquierda del texto de entrada en la ronda i-ésima.
- $R_{16}^{faulty} = R_{16} \oplus \text{error}$: mitad derecha de la entrada de la ronda 16 perturbada por un error inyectado.
- error : error inyectado.
- $P(\cdot)$: permutación a la salida de las cajas de sustitución.
- $S_j(\cdot)$: transformación de la caja de sustitución j-ésima.
- $PE_j(\cdot)$: permutación expansiva, que convierte los 32 bits de entrada en 48 bits que entran a las cajas de susitución.
- $K_{i,j}$: subclave de la caja de sustitución j-ésima de la ronda iésima.

El resultado de operar mediante XOR la salida correcta y la errónea tras anular la última permutación del algoritmo permite mostrar la información que el atacante necesita. La mitad derecha de dicha operación revela el error inyectado pudiendo así controlar los bits que se quieren inyectar mientras que la mitad izquierda muestra la diferencia XOR entre las salidas del texto encriptado correcta e incorrectamente al salir de la ronda 16. Dicha información aparece reflejada en la ecuación 3.3 donde se puede comprobar que la información referente a las rondas previas al fallo desaparece gracias a que dichas rondas encriptan los mismos datos para las dos encriptaciones (correcta y errónea). Por lo tanto es posible obtener la diferencia XOR entre las salidas de las cajas de sustitución de ambas encriptaciones. La ecuación 4 refleja dicha operación.

$$PI_j = S_j(PE_{16}(R_{16}) \oplus K_{16,j}) \oplus S_j(PE_{16}\left(R_{16}^{faulty}\right) \oplus K_{16,j} \qquad \text{para } j = 1,2,..8 \qquad (4)$$

Donde:

- PI_j : permutación inversa que anula la permutación tras la caja de sustitución j-ésima.

Es muy importante analizar la propagación de los bits de error de esta última ronda pues su efecto permitirá que se pueda descubrir la clave. Como el atacante conoce cuál va a ser la propagación de los bits a lo largo de la ronda, puede inyectar bits de error en posiciones que se propaguen a todas las cajas de sustitución. El objetivo de esto es producir una diferencia apreciable en las salidas de todas las cajas de sustitución ya que si no fuera el caso, aquellas cajas de sustitución que no reciban bits erróneos no se diferenciarán respecto a la encriptación válida original. Por lo tanto, en base a la ecuación 4 sólo es necesario generar claves parciales de 6 bits para cada caja de sustitución que conformarán toda la clave de la última ronda. Estas subclaves parciales son independientes unas de otras ya que actúan en caja de sustitución distintas.

Una vez se ha visto el esquema a seguir, el atacante sólo necesita emular la última ronda con cajas de sustitución idénticas para los dos casos (correcto y erróneo) de tal forma que probando con diferentes subclaves parciales para cada caja de sustitución, se deberá comprobar si se verifica la ecuación 4.

Como se ha visto, la idea básica consiste en emular la última ronda usando diversas subclaves parciales usando como verificación el valor de PI_j para un error previamente inyectado que afecte a todas las cajas de sustitución. Es muy importante que dicho error afecte a todas las cajas ya que si no fuera el caso, el atacante sólo podría obtener una parte de la clave de la última ronda. Sin embargo, si por cualquier caso el atacante quisiera afectar de manera aleatoria la última ronda, la clave final sólo se hallaría cuando todas las cajas se hayan activado. Dicho esto, una vez esté verificada la ecuación 4, el atacante obtiene una tabla de posibles candidatas a subclaves que han verificado la ecuación anterior. Por lo tanto ahora es necesario hacer un proceso de descarte, inyectando nuevos errores con el mismo u otro texto plano de entrada. La importancia de este último paso permite el poder descartar las subclaves parciales de 6 bits hasta quedar sólo una. Además como se comentó antes, el error deberá ser diferente en cada inyección, pero de eso se hablará en apartado 6 cuando se comprueben el resultado de las simulaciones.

Una vez la clave de la última ronda de 48 bits se ha calculado mediante inyecciones, el atacante deberá añadirle los otros 8 bits restantes que se habían descartado con anterioridad antes de acceder a dicha ronda, cuyo valor en principio es desconocido ya que no se han empleado en ninguna operación que refleje su valor. Esos bits restantes de la clave se deberán añadir a sus posiciones correspondientes de tal forma que el atacante pueda deshacer las rotaciones a las que se ha sometido la clave hasta llegar a la primera ronda. En dicha ronda la clave de 56 bits (48 bits conocidos + 8 bits desconocidos) se reordena invirtiendo la primera permutación hasta convertirla en la clave de entrada original. Esos 8 bits desconocidos se obtendrán haciendo una batería de simulaciones mediante fuerza bruta, probando un máximo de 256 posibles combinaciones hasta que la salida encriptada coincida con una salida de prueba.

5.1.2 Ataque en la ronda 15

El ataque visto anteriormente no sólo se limita a la última ronda, sino que se puede extender a la ronda 15 [32], donde la entrada fallida será $R_{15}^{faulty} = R_{15} \oplus error$. El error de este ataque se propaga según se puede ver en la figura 8. Matemáticamente al combinar la salida correcta (ec 5.1) con la errónea (ec 5.2) en la ecuación 5.3 aparece la dependencia de R_{15} que se puede anular conociendo el error, combinando la ecuación 5.3 con el error. Por lo tanto la efectividad de este ataque se basa en saber si se conoce o no el error inyectado. Dicho error a priori no se conoce, a menos que el atacante pueda inyectar determinados errores conocidos en cuyo caso se ahorraría al proceso que a continuación se va a explicar para poder deducir dicho error. Sin embargo lo normal a la hora de inyectar un error es que no se haga con la suficiente precisión como para poder saber el error que se ha inyectado. Por lo tanto lo primero que se debe hacer es determinar el error aislándolo en un conjunto de posibles valores. Para dicho propósito se puede usar la mitad derecha de la diferencia XOR entre las salidas correcta y errónea (ec 5.4).

$$C^{gold}: PF(F(k_{16}, R_{16}) \oplus L_{16}, R_{16}) \quad (5.1)$$

$$C^{faulty}: PF(F(k_{16}, F\left(k_{15}, R_{15}^{faulty}\right) \oplus L_{15}) \oplus \left(R_{15}^{faulty}\right), F\left(k_{15}, R_{15}^{faulty}\right) \oplus L_{15}) \quad (5.2)$$

$$PI\left(C^{gold} \oplus C^{faulty}\right) =$$

$$= (F(k_{16}, R_{16}) \oplus L_{16}, R_{16}) \oplus$$

$$\oplus F(k_{16}, F\left(k_{15}, R_{15}^{faulty}\right) \oplus L_{15}) \oplus \left(R_{15}^{faulty}\right), F\left(k_{15}, R_{15}^{faulty}\right) \oplus L_{15} \quad (5.3)$$

$$\Delta = F(k_{15}, R_{15}) \oplus F(k_{15}, R_{15} \oplus error) \quad (5.4)$$

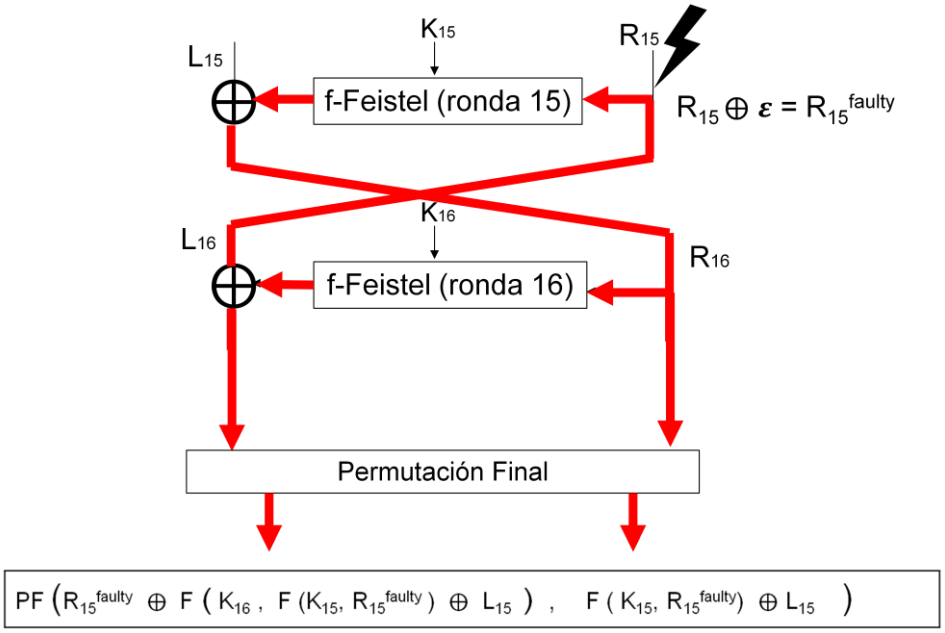


Figura 8. Efecto del error propagado en la penúltima ronda.

Con el fin de aislar la salida de la última ronda, es necesario aplicar la operación XOR a la mitad izquierda de la ecuación 5.3 con el error inyectado y teniendo en cuenta que $L_{16} = R_{15}$ para la encriptación correcta y que $L_{16} = R_{15}^{faulty}$ para la encriptación fallida, se obtendrá:

$$F(k_{16}, R_{16}) \oplus R_{15} \quad \oplus \quad F(k_{16}, F\left(k_{15}, R_{15}^{\text{faulty}}\right) \oplus L_{15}) \oplus R_{15}^{\text{faulty}} \oplus \text{error} =$$
$$F(k_{16}, R_{16}) \oplus F\left(k_{16}, F\left(k_{15}, R_{15}^{\text{faulty}}\right) \oplus L_{15}\right) =$$
$$S_j(PE_j(R_{16}) \oplus K_{16,j}) \oplus S_j(PE_j(R_{16}^{*}) \oplus K_{16,j}) = PI_j \text{ para } j = 1,2,..8 \qquad (6)$$

Donde PI_j es la permutación inversa de la última ronda, es decir la salida de las cajas de sustitución. De esta forma se consigue deshacer los últimos pasos de permutación gracias a que el atacante conoce el error.

Como se puede comprobar la dificultad de este método consiste en averiguar el error inyectado ya que este error permitirá al atacante anular la dependencia con las rondas previas. Así usando el error se podrá generar la salida que servirá como referencia para hacer las sucesivas pruebas hasta producir coincidencias parciales en cada caja de sustitución consiguiendo aislar un conjunto de subclaves candidatas que satisfagan la ecuación 6.

Tal y como se ha explicado en el apartado anterior (en el ataque a la última ronda) habrá que repetir el proceso anterior generando subclaves parciales para cada caja de sustitución, cosa que se puede hacer con un contador de 6 bits que afecte a todas las cajas.

Como se ha podido observar el proceso para generar las claves es el mismo que en el apartado anterior con la excepción de que ahora es necesario conocer el error cometido para poder deshacer los últimos pasos de permutación, cosa que antes no lo era pues al estar el error en la última ronda, la operación XOR conseguía eliminar la dependencia con las rondas previas al ataque.

5.1.3 Extensión del ataque a otras rondas

La extensión del modelo de fallos anteriormente explicado también se puede aplicar a otras rondas. Sin embargo dado que ese no es el principal objetivo del proyecto no será explicado aunque se deja al lector la referencia bibliográfica [32] donde se explican dichos ataques.

5.2 Conclusiones

Se ha estudiado un ataque de tipo DFA sobre el algoritmo DES y se ha comprobado de forma teórica sus efectos en la ronda 16 tomando como base el desarrollo matemático que sustenta el algoritmo. Los resultados teóricos muestran la posibilidad de obtener la clave a través de la inyección de errores en la última ronda gracias a los cuales el atacante puede inferir información en la salida del algoritmo. Dicha información muestra además del error inyectado (gracias al cual el atacante puede controlar la inyección) la salida de la última ronda (la cual se usa para deshacer los últimos pasos de encriptación) con lo que el atacante podrá usar para

comparar cuando emule la última ronda con bits de prueba como subclaves hasta producir coincidencias parciales. Esto además se ha podido extender a la ronda 15 con una dificultad añadida pues se necesita conocer el error inyectado para poder anular la dependencia con las otras rondas.

6. Reproducción del ataque de fallos en simulación

Se ha visto en el capítulo anterior cómo un atacante puede inyectar un fallo en una determinada ronda produciendo un resultado apreciable a la salida, a partir del cual se puede inferir información crucial para obtener la clave.

Este capítulo presenta la reproducción del error inyectado como una primera aproximación usando el simulador ISIM de Xilinx, instrumentalizando el código en VHDL. La idea es usar las simulaciones para determinar la viabilidad de reproducir estos ataques en hardware, es decir, primero ver si es posible reproducir los fallos en simulación para luego pasar a la implementación hardware. Para ello se llevarán a cabo una serie de inyecciones completamente automáticas que luego se usarán para ser post-procesadas obteniendo las candidatas a subclaves hasta obtener finalmente la clave de la última ronda.

6.1 Reproducción forzando señales en el mismo simulador ISIM de Xilinx

En una primera aproximación para comprender los efectos de un error propagado, se empleará el simulador ISIM (ISE Simulator) de Xilinx pues ofrece la posibilidad de forzar valores en determinadas señales (sean o no registros) pudiendo observar el efecto de propagación de dicho cambio. Gracias a este forzado de señales el atacante puede ser capaz de alterar cualquier registro haciendo que tome el valor que desee. En este caso, es suficiente que el error inyectado afecte a todas las cajas de sustitución. Por lo tanto cualquiera que tenga acceso al código VHDL del DES puede forzar señales y comprobar a través de los cronogramas su propagación. Esto permite visualizar el ataque desde un punto de vista más práctico.

Para reproducir los fallos de manera virtual es necesario:

- El algoritmo DES implementado en VHDL [34]. Este algoritmo DES permite encriptar un bloque de 64 bits usando una clave de 64 bits (que luego es reducida al desechar 8 bits, quedando 56 bits). Además al ser un cifrado en bloque, el algoritmo permite que operar en modo de encriptación o desencriptación empleando el mismo código.
- La herramienta Xilinx-ISE (Integrated Software Environment), una herramienta de diseño de circuitos profesional que opera con lenguajes HDL tales como VHDL o Verilog y que permite:
    - Realizar diseños hardware (sintetizando los diseños).
    - Estimular las entradas de los diseños.
    - Simular los diseños

  Esta herramienta se apoya sobre:

    - Entorno ISE: el cual realiza el diseño del circuito a partir de un esquemático o de un lenguaje de diseño hardware como VHDL o Verilog.
    - Entorno ISIM (ISE Simulator): el cual se encarga de simular el código en base a unos estímulos previamente asignados en un archivo aparte, una vez que el diseño ha compilado correctamente en el entorno ISE.

Para llevar a cabo el forzado de señales es necesario:

- Compilar todo el código y simularlo mediante un test-bench con unos estímulos dados.
- Situarse en el instante donde se quiere inyectar el fallo usando como referencia el registro contador de ronda.
- Situarse en el registro donde se almacena el texto de entrada a la ronda y sustituir el valor de dicho registro por otro. El valor resultante debe corresponderse con un posible error inyectado.

La figura 9 muestra una captura de un error inyectado mostrando el efecto de la propagación del cambio en el registro. En dicha figura el registro "inmsg" es forzado resultando: 0x4346cd98286c0b03 xor 0xb9aa9496 = 0x5346cd9891c69f95.

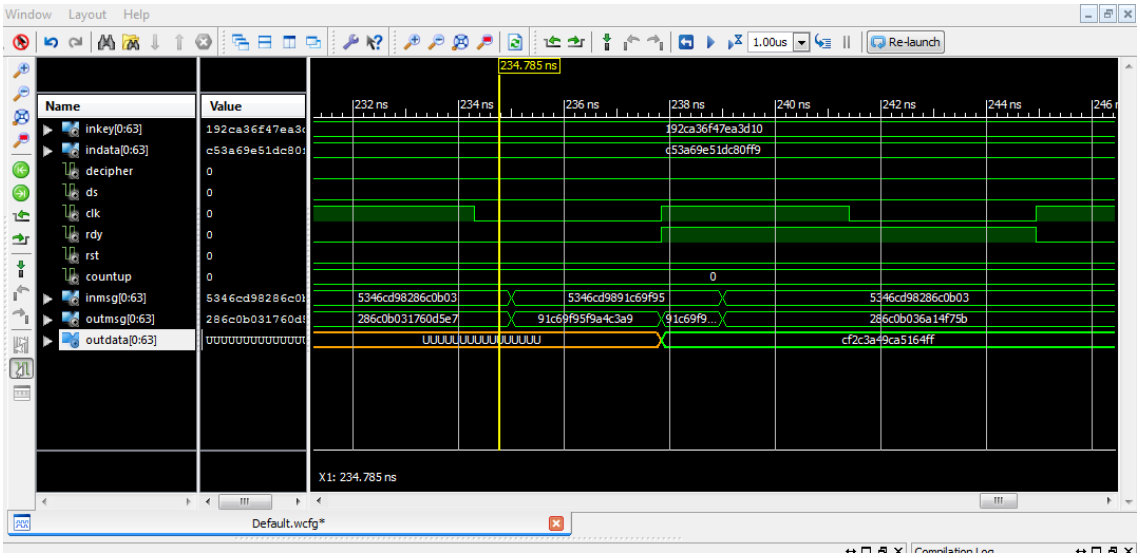


Figura 9. Forzado del registro "inmsg" mediante el simulador ISIM de Xilinx.

6.2 Reproducción instrumentalizada del código en VHDL y obtención de la clave.

Anteriormente se ha visto y comprendido cómo puede afectar la inyección y propagación de un error en un determinado registro. Así que ahora se instrumentalizará el código para comprobar la viabilidad del ataque propuesto. Para ello se reproducirá en VHDL el ataque propuesto automatizando todos los procesos que llevan a la obtención de la clave. Esta reproducción requiere del uso de componentes adicionales al bloque que implementa DES. A continuación se enumeran los componentes necesarios que además se muestran en la figura 10:

1. **Generador de textos planos (opcional)**
   Este primer bloque genera textos planos a partir de uno de entrada, para hacer múltiples inyecciones. De todas formas este bloque es opcional pues tal y como se mostrará en los resultados cambiar de textos planos no es determinante para encontrar la clave siempre y cuando los errores sean distintos. Esto se explicará en el apartado de resultados donde se abordarán todas las posibilidades de funcionamiento.

2. **Inyector de errores**
   Con el texto plano generado anteriormente, se ejecuta el algoritmo DES y posteriormente se inyecta el error sobre una serie de bits que afecten a todas las cajas de sustitución. Este inyector de errores está programado para cambiar los bits de error en sucesivos ataques permitiendo generar distintas parejas de textos cifrados con y sin error. De lo contrario el atacante no podría reducir la lista de subclaves candidatas debido a que el error inyectado produce una incertidumbre sobre los bits de error. Por ejemplo si el atacante atacara siempre con el mismo error (aunque el texto plano de entrada cambie) el número de subclaves candidatas en cada caja de sustitución será de 2. Este hecho es muy importante y se valorará en el apartado de resultados. Un ejemplo de inyección es mostrado en la figura 11.

3. **Post-procesado**
   Una vez el atacante inyecta un error, el bloque del post-procesado deshace las últimas permutaciones en base a la información de salida del texto encriptado correcta e incorrectamente, llegando justo a la salida de las cajas de sustitución dentro de la última ronda. De esta forma se deberá emular la funcionalidad de la última ronda para ambos casos (correcto y erróneo). Así sólo quedará probar claves de 6 bits en cada caja. Esto se puede llevar a cabo usando un único contador de 6 bits que reutilicen todas las cajas en paralelo.
   Una vez las candidatas a subclave han sido generadas, este proceso se comunica con el generador de textos planos para generar otra entrada a encriptar. Luego se inyecta un fallo distinto al anterior, consiguiendo otra lista de candidatas gracias a la cual se podrán descartar aquellas que no coincidan.

4. **Despermutación de subclave**
   Una vez la subclave de la última ronda ha sido obtenida gracias a las inyecciones y continuos post-procesados se le añaden los 8 bits que faltan y que no se usan en dicha ronda. Estos 8 bits al no usarse no se pueden deducir directamente por lo que en un principio se les asignará un valor de ‘U’ (uninitialized). A continuación se deshacen las rotaciones a las que se somete a la subclave hasta llegar al principio del algoritmo donde aún faltan esos 8 bits sin valores asignados.

5. **Generador de 8 bits**
   Ahora sólo queda hacer un pequeño ataque de fuerza bruta con 8 bits de clave. Esto es así ya que el ataque únicamente permitía deducir los 48 bits usados en la ronda final. Sin embargo no supone un esfuerzo desmedido pues con un máximo de 256 simulaciones se puede romper el cifrado.

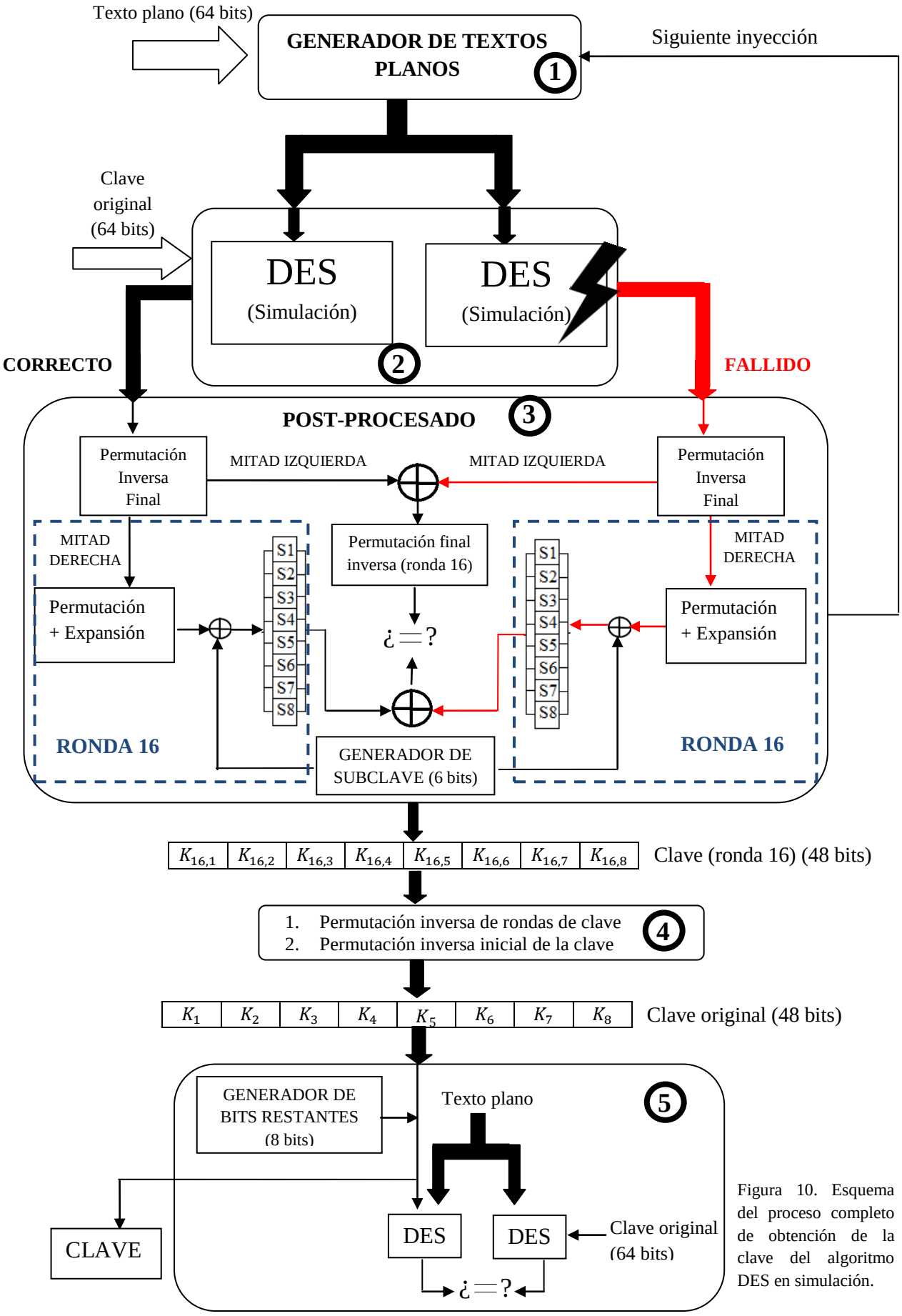


Figura 10. Esquema del proceso completo de obtención de la clave del algoritmo DES en simulación.

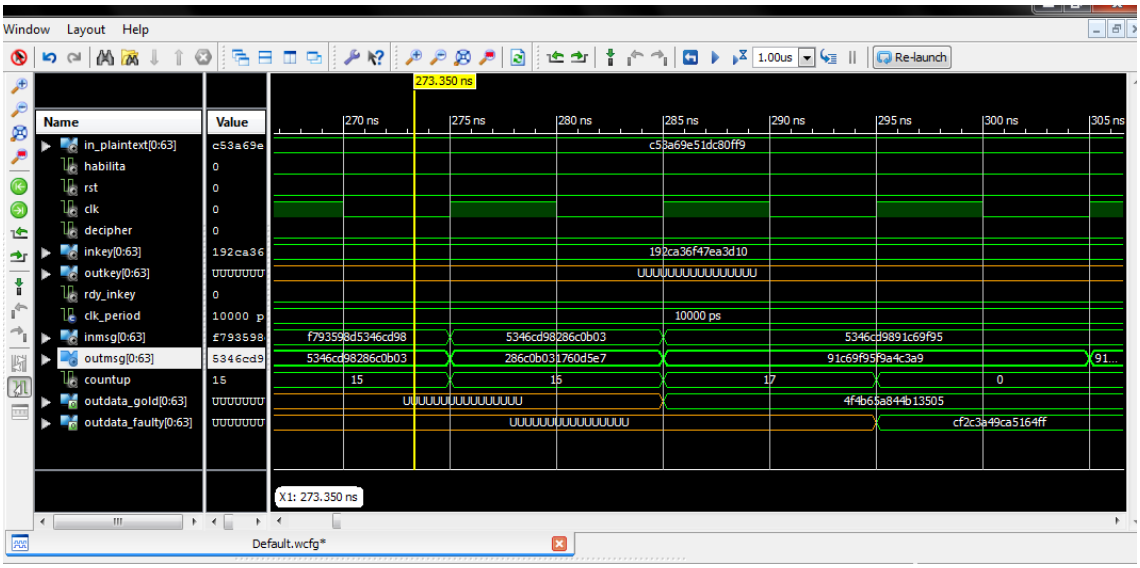


Figura 11. Ejemplo del inyector de errores del modelo instrumentalizado en VHDL.

6.3 Resultados y experiencias

El análisis de este ataque permite conocer mejor cómo un algoritmo criptográfico puede resultar desprotegido frente a este tipo de ataque. Dado que el algoritmo es conocido, su estudio permite conocer su comportamiento frente a diversas situaciones permitiendo que alguien pueda aprovechar ciertas vulnerabilidades.

La instrumentalización de todo el proceso permite que el atacante pueda comprobar la viabilidad de todo el proceso ya desde el punto de vista más práctico permitiéndole saber el número medio de inyecciones, el tipo de error en cada inyección y la necesidad o no de recurrir a distintos textos planos en cada ataque.

En primer lugar para conocer la efectividad del ataque es necesario realizar un estudio que permita conocer el número medio de ataques necesarios. Esto a priori no se sabe, sin embargo y gracias a la instrumentalización del ataque en VHDL se puede averiguar programando un tipo de ataque en cada inyección. Dicho error en cada ataque debe cumplir la condición de que active todas las cajas de sustitución de la última ronda para poder efectuar una comparación parcial de la salida de dichas cajas. Para hacerlo es suficiente con cambiar un solo bit que entre en la caja que el atacante quiere activar, para poder producir una salida diferente a la salida encriptada de forma correcta. La tabla 2 muestra los bits de entrada en cada caja de sustitución procedentes del registro llamado “inmsg” donde el atacante se centrará en atacar. Dicha tabla proporciona al atacante las posiciones donde se pueden inyectar bits de error siempre y cuando afecten a todas las cajas de sustitución. Como se puede comprobar en dicha tabla, el registro “inmsg” se expande en cada ronda por lo que habrá bits que se compartirán en como mucho dos cajas. Esto permite que los errores inyectados de forma aleatoria faciliten la activación de las cajas de sustitución.

S-Box	registro: inmsg (bit i-ésimo)					
S1	63	36	32	33	34	35
S2	35	40	36	37	38	39
S3	39	44	40	41	42	43
S4	43	48	44	45	46	47
S5	47	52	48	49	50	51
S6	51	56	52	53	54	55
S7	55	60	56	57	58	59
S8	59	32	60	61	62	63

Tabla 2. Relación de los bits del registro inmsg con cada caja de sustitución.

La figura 12 muestra un ejemplo de la propagación de un error inyectado en los bits: 33, 40, 43, 50, 56, 59 del registro "inmsg" de la última ronda. Dicho error se traduce en una operación XOR sobre la mitad derecha del registro "inmsg", es decir, inmsg(32 to 63) XOR x"40902090". Como se puede observar en la misma figura, se cumple la condición de que al menos un bit ha sido alterado en la entrada de cada caja de sustitución.

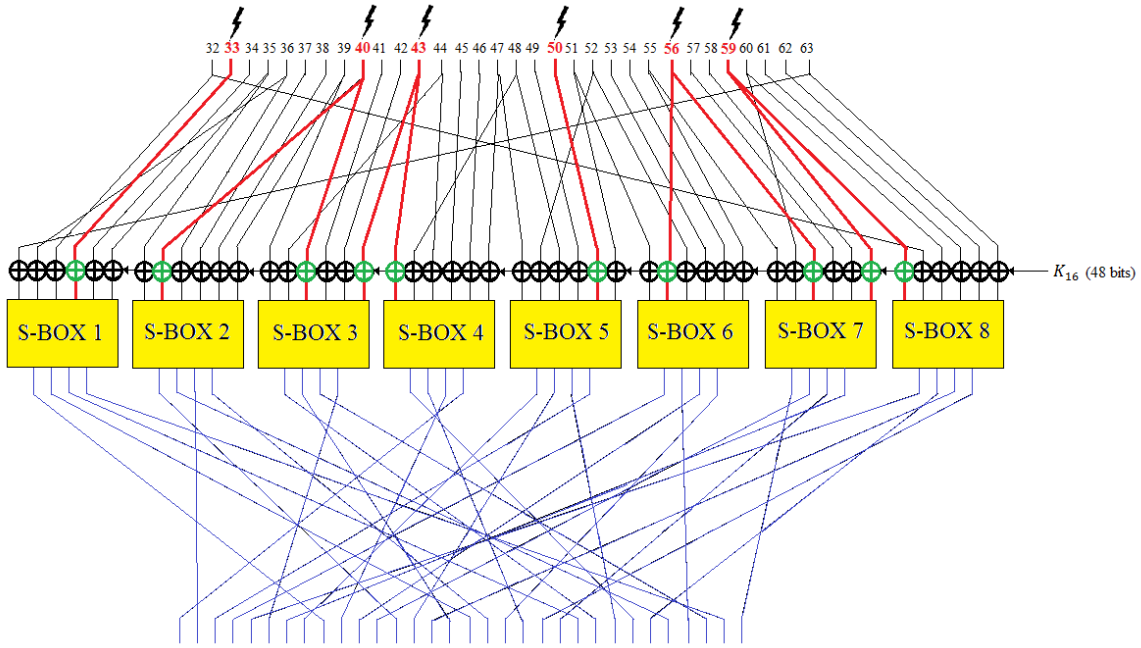


Figura 12. Propagación detallada de un error inyectado en los bits: 33, 40, 43, 50, 56, 59 del registro "inmsg"

Cuando el error ha sido inyectado se obtienen subclaves candidatas correspondientes a cada caja de sustitución. Estas candidatas guardan entre sí una relación referente a los bits de error. Por ejemplo si el atacante decide inyectar un error sobre alguna caja en un determinado bit de entrada, las candidatas se repiten de dos en dos con ese bit asignado con todos sus posibles valores. De la misma forma si se inyectaran dos errores en la entrada de alguna caja, las

candidatas se podrán agrupar en subconjuntos compuestos por un grupo de bits en común y diferentes variaciones en los bits de error. Este hecho se puede comprobar en la tabla 3, donde se analiza un ejemplo usando para ello un texto plano y clave cualquiera, y el mismo error inyectado que en la figura 11.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
00	04	04	00	01	0d	02	07
04	0f	0d	01	03	1d	05	0a
12	14	14	04	20		0b	0c
16	1f	1d	0a	22		0c	10
31	23	24	19	2d		10	15
35	2b	25	1c	2f		14	19
	33	26	20			19	1d
	3b	27	21			1d	27
		2c	24				2a
		2d	2a				2c
		2e	39				30
		2f	3c				35
		31					39
		38					3d

Tabla 3. Lista de subclaves candidatas para el texto plano: 0xc53a69e51dc80ff9 , clave: 0x192ca36f47ea3d10 , error: 0x40902090.

Sabiendo el error de cada caja de sustitución se pueden aislar los subgrupos de subclaves candidatas. En el conjunto de tablas: 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8 aparecen reflejadas los distintos subgrupos con el/los bit/s de error.

Error: 000100	
$K_{16,1}(hex)$	$K_{16,1}(bin)$
00	000000
04	000100
12	010010
16	010110
31	110001
35	110101

Tabla 4.1 Subgrupos de la subclave de la caja 1

Error: 010000	
$K_{16,2}(hex)$	$K_{16,2}(bin)$
04	000100
14	010100
0f	001111
1f	011111
23	100011
33	110011
2b	101011
3b	111011

Tabla 4.2 Subgrupos de la subclave de la caja 2

Error: 001001	
$K_{16,3}(hex)$	$K_{16,3}(bin)$
04	000100
0d	001101
14	010100
1d	011101
24	100100
25	100101
2c	101100
2d	101101
26	100110
27	100111
2e	101110
2f	101111
31	110001
38	111000

Tabla 4.3 Subgrupos de la subclave de la caja 3

Error: 100000	
$K_{16,4}(hex)$	$K_{16,4}(bin)$
00	000000
20	100000
04	000100
24	100100
19	011001
39	111001
01	000001
21	100001
0a	001010
2a	101010
1c	011100
3c	111100

Tabla 4.4 Subgrupos de la subclave de la caja 4

Error: 000010	
$K_{16,5}(hex)$	$K_{16,5}(bin)$
01	000001
03	000011
20	100000
22	100010
2d	101101
2f	101111

Tabla 4.5 Subgrupos de la subclave de la caja 5

Error: 010000	
$K_{16,6}$(hex)	$K_{16,6}$(bin)
0d	001101
1d	011101

Tabla 4.6 Subgrupos de la subclave de la caja 6

Error: 001001	
$K_{16,7}(hex)$	$K_{16,7}(bin)$
02	000010
0b	001011
05	000101
0c	001100
10	010000
19	011001
14	010100
1d	011101

Tabla 4.7 Subgrupos de la subclave de la caja 7

Error: 100000	
$K_{16,8}(hex)$	$K_{16,8}(bin)$
07	000111
27	100111
0c	001100
2c	101100
15	010101
35	110101
1d	011101
3d	111101
2a	101010
0a	001010
30	110000
10	010000
39	111001
19	011001

Tabla 4.8 Subgrupos de la subclave de la caja 8

Sin embargo un solo ataque no es suficiente por lo que será necesario otro para poder reducir la lista de subclaves candidatas repitiendo el proceso anterior de inyección. Este proceso interativo se puede plantear de dos formas:

1. Usar mismo texto plano en cada ataque.

    1.1 Inyectar el mismo error en cada ataque. En este método no se obtienen distintas parejas (C^{gold}, C^{faulty}) con lo que no se puede reducir la lista de subclaves candidatas y por lo tanto no se podrá obtener la clave original.

    1.2 Inyectar distintos errores en cada ataque. De esta forma con cada pareja distinta se reduce la lista de subclaves candidatas hasta quedar finalmente una, siempre y cuando los errores sean distintos, incluyendo los errores parciales en cada caja de sustitución. Así se podrá obtener la clave original. Las tablas 5.1, 5.2, 5.4 y 5.6 muestran varios ejemplos de diferentes subclaves candidatas.

$\boldsymbol{Key_{16}(hex)}$							
$\boldsymbol{K_{16,1}}$	$\boldsymbol{K_{16,2}}$	$\boldsymbol{K_{16,3}}$	$\boldsymbol{K_{16,4}}$	$\boldsymbol{K_{16,5}}$	$\boldsymbol{K_{16,6}}$	$\boldsymbol{K_{16,7}}$	$\boldsymbol{K_{16,8}}$
04	10	00	01	22	05	02	06
1f	14	03	09	24	0a	05	07
2a	1a	17	13	2b	0c	0b	18
2b	23	1a	1b	2d	0d	0c	2e
30	29	1b	24		31	10	30
31	2d	20	2c		38	14	31
		21	36		39	19	
		2d	3e		3e	1d	
		39					
		3a					

Tabla 5.1 Subclaves candidatas para texto plano: 0xc53a69e51dc80ff9, clave: 0x192ca36f47ea3d10, error: 0xb9aa9496.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	07	00	01	09	0d	01	20
07	16	17	09	2d	18	0b	21
08	23	1f	0a			10	26
0b	32	2d	0f			12	27
19		32	11			21	30
1a		3a	19			2b	36
20			1a			38	
23			1f			3a	
			26				
			28				
			2d				
			2e				
			36				
			38				
			3d				
			3e				

Tabla 5.2 Subclaves candidatas para texto plano: 0xc53a69e51dc80ff9, clave: 0x192ca36f47ea3d10, error: 0x35d745a6.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	00	01	2d	0d	0b	30
		17	09			10	
		2d	36				
		3a	3e				

Tabla 5.3. Coincidencias parciales entre las tablas 5.1 y 5.2.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	05	07	01	0c	0d	00	0e
06	08	0d	04	2d	2b	05	15
1a	0b	24	08			0b	23
24	0d	27	0d			0d	2b
38	11	2d	21			10	30
3a	12	2e	24			15	38
	17					1b	
	1f					1d	
	23					27	
	39					2c	
						2f	
						37	
						3c	
						3f	

Tabla 5.4 Subclaves candidatas para texto plano: 0xc53a69e51dc80ff9, clave: 0x192ca36f47ea3d10, error: 0xeaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	2d	01	2d	0d	0b	30
						10	

Tabla 5.5. Coincidencias parciales entre las tablas 5.3 y 5.4.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
02	07	12	01	22	0d	04	04
04	10	15	17	24	11	07	1a
06	16	1b	1f	2b	21	08	2e
09	23	1c	23	2d	3d	0b	30
0b	32	29	2b			15	
0b	34	2a	35			1a	
0f		2d				26	
20		2e				29	
2d		3b				32	
		3c				37	
						38	
						3d	

Tabla 5.6. Subclaves candidatas para texto plano: 0xc53a69e51dc80ff9, clave: 0x192ca36f47ea3d10, error: 0xd47490f4.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	2d	01	2d	38	0b	30

Tabla 5.7. Coincidencias parciales entre las tablas 5.5 y 5.6.

Como se ha podido observar han hecho falta 4 inyecciones distintas aunque habrían sido más si se hubieran considerado los resultados de la Tabla 3 en lugar de la tabla 4.4, ya que los resultados de esta Tabla 3 no aportan información vital de descarte. Esto es debido a que el error propagado de la Tabla 3 en la caja 7 es el mismo que el de la tabla 5.2.

2. Usar distintos textos planos en cada ataque.

   2.1 Inyectar el mismo error en cada ataque. Se obtienen distintas parejas (C^{gold}, C^{faulty}) sólo como consecuencia de usar distintos textos planos y se reduce la lista de subclaves candidatas hasta llegar a un punto en el cual no se puede reducir más. Esto se debe a que cuando se lleva a cabo un ataque, las coincidencias parciales se agrupan en subgrupos de candidatas. Sin embargo al repetir el mismo error en los

sucesivos ataques se conseguirá discriminar varios subgrupos hasta quedar dos posibilidades. Las tablas: 6.1 a 6.15 muestran el proceso de descarte de subclaves parciales hasta llegar a un punto de no reducción.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	05	07	01	0c	0d	00	0e
06	08	0d	04	2d	2b	05	15
1a	0b	24	08			0b	23
24	0d	27	0d			0d	2b
38	11	2d	21			10	30
3a	12	2e	24			15	38
	17					1b	
	1f					1d	
	23					27	
	39					2c	
						2f	
						37	
						3c	
						3f	

Tabla 6.1 Subclaves candidatas para texto plano: 0xc53a69e51dc80ff9, clave: 0x192ca36f47ea3d10, error: 0xeaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	17	01	0c	0d	03	26
1b	39	1d	04	2d	2b	0b	2b
25		25	0a			13	30
3a		27	0f			1b	3d
		2d	21				
		2f	24				

Tabla 6.2 Subclaves candidatas para texto plano: 0xe29d34f28ee407f, clave: 0x192ca36f47ea3d10, error: 0xeaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	27	01	0c	0d	0b	2b
3a	39	2d	04	2d	2b	1b	30
			21				
			24				

Tabla 6.3. Coincidencias parciales entre las tablas 6.1 y 6.2.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	17	01	01	0d	03	20
1a	39	1d	04	02	0e	05	27
1b		25	21	07	28	0b	2b
23		27	24	0c	2b	0e	30
24		2d	2a	11		13	3b
25		2f	2f	13		15	3c
3a				14		1b	
				20		1e	
				23		21	
				26		22	
				2d		29	
				30		31	
				32		32	
				35		39	

Tabla 6.4 Subclaves candidatas para texto plano: 0x714e9a79477203f3, clave: 0x192ca36f47ea3d10, error: 0xeaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	27	01	0c	0d	0b	2b
3a	39	2d	04	2d	2b	1b	30
			21				
			24				

Tabla 6.5. Coincidencias parciales entre las tablas 6.3 y 6.4.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	16	01	09	0d	00	27
1b	39	1c	04	0c	2b	05	2b
25		21	0a	11		0b	2c
3a		27	0f	1a		0d	30
		2b	21	1c		10	37
		2d	24	1f		15	3c
				28		1b	
				2d		1d	
				2f		27	
				30		2c	
				3d		2f	
				3e		37	
						3c	
						3f	

Tabla 6.6 Subclaves candidatas para texto plano: 0x38a74d3ca3b901ff, clave: 0x192ca36f47ea3d10, error: 0x eaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	27	01	0c	0d	0b	2b
3a	39	2d	04	2d	2b	1b	30
			21				
			24				

Tabla 6.7. Coincidencias parciales entre las tablas 6.5 y 6.6.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	23	01	02	0a	03	0e
06	39	27	04	09	0b	08	15
38		29	08	0c	0d	0b	23
3a		2d	0d	0f	18	13	2b
		33	21	1a	2b	18	30
		39	24	1d	2c	1b	38
				1f	2d	21	
				23	3e	24	
				28		29	
				2d		2f	
				2e		31	
				3b		34	
				3c		39	
				3e		3f	

Tabla 6.8 Subclaves candidatas para texto plano: 0x9c53a69e51dc80ff, clave: 0x192ca36f47ea3d10, error: 0x eaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	27	01	0c	0d	0b	2b
3a	39	2d	04	2d	2b	1b	30
			21				
			24				

Tabla 6.9. Coincidencias parciales entre las tablas 6.7 y 6.8.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	03	00	01	00	05	08	26
06	19	06	04	03	0d	09	2b
1a	23	0a	21	09	1d	0b	2e
24	24	0c	24	0c	1e	0e	30
38	27	27	2a	13	23	18	35
3a	39	2d	2f	14	2b	19	3d
	3d	35		21	38	1b	
	3e	3f		22	3b	1d	
				28		1e	
				2d		2c	
				32		3c	
				35			

Tabla 6.10. Subclaves candidatas para texto plano: 0xce29d34f28ee407f, clave: 0x192ca36f47ea3d10, error: 0xeaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	27	01	0c	0d	0b	2b
3a	39	2d	04	2d	2b	1b	30
			21				
			24				

Tabla 6.11. Coincidencias parciales entre las tablas 6.9 y 6.10.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	27	01	01	0d	01	20
1b	39	2d	04	0c	2b	04	27
25			0a	11		0b	2b
3a			0f	1d		11	30
			21	1e		14	3b
			24	20		1b	3c
				2d			
				30			
				3c			
				3f			

Tabla 6.12. Subclaves candidatas para texto plano: 0xe714e9a79477203f, clave: 0x192ca36f47ea3d10, error: 0xeaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	27	01	0c	0d	0b	2b
3a	39	2d	04	2d	2b	1b	30
			21				
			24				

Tabla 6.13. Coincidencias parciales entre las tablas 6.11 y 6.12.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
00	05	07	01	03	09	00	2b
04	1f	0d	04	06	0c	06	30
06	23	27	33	0c	0d	0b	
0a	2b	2d	36	0f	2a	0e	
11	2d			1b	2b	10	
14	2e			1c	2f	16	
2a	31			22		1b	
2f	34			27		1e	
34	37			2d		24	
38	39			2e		27	
3a				3a		2c	
3e				3d		34	
						37	
						3c	

Tabla 6.14. Subclaves candidatas para texto plano: 0xf38a74d3ca3b901f, clave: 0x192ca36f47ea3d10, error: 0xeaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	27	01	0c	0d	0b	2b
3a	39	2d	04	2d	2b	1b	30

Tabla 6.15. Coincidencias parciales entre las tablas 6.13 y 6.14.

Como se ha podido observar en las tablas anteriores, las dos primeras tablas correspondientes a las dos primeras inyecciones permiten simplificar bastante la tabla de candidatas, sin embargo los sucesivos intentos de atacar el algoritmo no permiten obtener nueva información hasta el octavo ataque donde se reduce un poco más la tabla de candidatas. Sin embargo a partir de este punto no se pueden reducir más las coincidencias parciales. Este lento proceso de descarte incompleto se ha podido llevar a cabo bajo la condición del cambio de textos planos ya que junto al ataque producirán distintas parejas de textos encriptados aunque se ataque con el mismo error. Aun así, se llega irremediablemente a un punto de no avance llegando a tener dos alternativas por

cada caja de sustitución. No obstante el número de posibles claves de la última ronda equivale a $(2 \text{ posibilidades})^{8 \text{ cajas}} = 256 \text{ posibilidades}$, por lo que sería posible llevar a cabo un barrido de comprobaciones para cada una de estas posibilidades, aunque no se sabrá si la clave probada es la correcta hasta que el ataque de fuerza bruta (8 bits) final muestre una muestre una coincidencia completa y única entre dos textos encriptados con la clave de prueba y la original, lo que implica que es necesario obtener esos 16 bits a la vez por lo que si quieren obtener por fuerza bruta son 2^{16} = 65536 posibilidades.

2.2 Inyectar un error distinto en cada ataque respecto al anterior. Esto quiere decir que se pueden repetir los ataques siempre y cuando sean distintos respecto al anterior ataque. De esta forma se puede optar por usar como mínimo 2 errores intercalándolos uno tras otro siempre. La condición fundamental es que sean distintos en cada caja de sustitución respecto al anterior ataque. Del mismo modo se pueden emplear más ataques distintos lo que reducirá el número de inyecciones. Las pruebas llevadas a cabo muestran que con 2 errores distintos la clave de la última ronda en 4 inyecciones. Las tablas: 7.1 a 7.7 reflejan el proceso de obtención de dicha clave para el caso de usar 2 errores.

$\boldsymbol{Key_{16}(hex)}$							
$\boldsymbol{K_{16,1}}$	$\boldsymbol{K_{16,2}}$	$\boldsymbol{K_{16,3}}$	$\boldsymbol{K_{16,4}}$	$\boldsymbol{K_{16,5}}$	$\boldsymbol{K_{16,6}}$	$\boldsymbol{K_{16,7}}$	$\boldsymbol{K_{16,8}}$
04	23	01	00	14	0d	01	0e
0a	31	0d	01	1b	13	0b	1a
16		17	12	22		21	24
18		1b	13	2d		2b	30
36		21	20			31	
38		2d	21			34	
			28			3b	
			29			3e	

Tabla 7.1. Subclaves candidatas para texto plano: 0xc53a69e51dc80ff9, clave: 0x192ca36f47ea3d10, error: 0x54cae96b.

$\boldsymbol{Key_{16}(hex)}$							
$\boldsymbol{K_{16,1}}$	$\boldsymbol{K_{16,2}}$	$\boldsymbol{K_{16,3}}$	$\boldsymbol{K_{16,4}}$	$\boldsymbol{K_{16,5}}$	$\boldsymbol{K_{16,6}}$	$\boldsymbol{K_{16,7}}$	$\boldsymbol{K_{16,8}}$
04	10	05	01	08	04	0b	11
0a	17	19	09	16	0d	11	13
0f	23	2d	13	25	30	27	18
21	24	31	1b	2d	31	3d	1a
2a			24	33	38		30
2f			2c	3b	39		36
			36				3b
			3e				3d

Tabla 7.2. Subclaves candidatas para texto plano: 0xe29d34f28ee407fc, clave: 0x192ca36f47ea3d10, error: 0x5e2c16ea4.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	2d	01	2d	0d	0b	1a
0a			13				30

Tabla 7.3. Coincidencias parciales entre las tablas 7.1 y 7.2.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	21	00	07	00	01	01
0a	25	2d	01	1b	0d	06	07
17	31		0e	2d	13	0b	13
19	37		0f	31	1e	0c	15
22			10			21	24
23			11			26	2e
2c			1e			2b	30
2d			1f			2c	3a
33							
3d							

Tabla 7.4. Subclaves candidatas para texto plano: 0x714e9a79477203fe, clave: 0x192ca36f47ea3d10, error: 0x54cae96b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	2d	01	2d	0d	0b	30
0a							

Tabla 7.5. Coincidencias parciales entre las tablas 7.3 y 7.4.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	07	07	01	00	01	0a	10
21	08	1b	09	01	04	0b	13
	0b	24	1b	1e	05	0d	15
	17	2b	24	1f	08	1c	18
	23	2d	2c	25	0c	2a	1b
	33	31	36	2d	0d	3c	1e
	3c	37	3e	33	10	3d	30
	3f	38		3b	19		32
					35		35
					3c		39
							3b
							3e

Tabla 7.6. Subclaves candidatas para texto plano: 0x38a74d3ca3b901ff, clave: 0x192ca36f47ea3d10, error: 0x5e2c16ea4.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	2d	01	2d	0d	0b	30

Tabla 7.7. Coincidencias parciales entre las tablas 7.5 y 7.6.

Como se ha podido observar en las tablas anteriores ha bastado con 4 inyecciones de distintos textos planos y dos errores diferentes intercalados entre sí, para obtener las subclaves de la última ronda.
Como se ha comentado con anterioridad, aumentar el número de errores diferentes en cada inyección puede provocar una disminución del número de inyecciones. Eso lo demuestra la tabla 8.1 donde al combinar dichos resultados con los de la tabla 7.3 (candidatas por medio de 2 errores diferentes) se puede determinar la clave con sólo 3 inyecciones de 3 errores diferentes. Para ello ha sido de vital importancia que los errores fueran diferentes, pero además que mostraran una diferencia notable entre sí como por ejemplo alternar los bits de error para no repetirlos en sucesivos ataques.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
02	00	07	01	07	0b	04	10
03	01	1c	1d	14	0c	0b	15
04	05	2d	2d	26	0d	21	30
06	09	36	31	2d	1e	2e	35
0b	13			35	22	32	
12	14			3e	30	3d	
13	19				31		
15	1d				37		
17	23						
1a	27						
22	2a						
2c	2d						
33	37						
3d	3b						
	3e						
	3f						

Tabla 8.1. Subclaves candidatas para texto plano: 0x714e9a79477203fe, clave: 0x192ca36f47ea3d10, error: 0x1ebc3cf0.

Este método permite además reducir el número de inyecciones respecto al caso de usar el mismo texto plano con diferentes errores. Las tablas 8.1 a 8.4 muestran que usando diferentes textos planos con los mismos errores que se usaron para el caso de mantener el mismo texto plano se consigue reducir el número de inyecciones.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	10	00	01	22	05	02	06
1f	14	03	09	24	0a	05	07
2a	1a	17	13	2b	0c	0b	18
2b	23	1a	1b	2d	0d	0c	2e
30	29	1b	24		31	10	30
31	2d	20	2c		38	14	31
		21	36		39	19	
		2d	3e		3e	1d	
		39					
		3a					

Tabla 9.1. Subclaves candidatas para texto plano: 0x0xc53a69e51dc80ff9, clave: 0x192ca36f47ea3d10, error: 0xb9aa9496.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	07	00	01	01	03	09	10
07	16	03	02	09	0a	0b	16
	23	2d	07	0f	0d	12	30
	32	2e	09	25	16	18	36
			11	2b	18	21	3b
			12	2d	1f	23	3d
			17		29	32	
			19		3c	38	
			20				
			25				
			26				
			2e				
			30				
			35				
			36				
			3e				

Tabla 9.2. Subclaves candidatas para texto plano: 0x0xe29d34f28ee407fc, clave: 0x192ca36f47ea3d10, error: 0x35d745a6.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	00	01	2b	0a	0b	30
		03	09	2d	0d		
		2d	36				
			3e				

Tabla 9.3. Coincidencias parciales entre las tablas 9.1 y 9.2.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	17	01	01	0d	03	20
1a	39	1d	04	02	0e	05	27
1b		25	21	07	28	0b	2b
23		27	24	0c	2b	0e	30
24		2d	2a	11		13	3b
25		2f	2f	13		15	3c
3a				14		1b	
				20		1e	
				23		21	
				26		22	
				2d		29	
				30		31	
				32		32	
				35		39	

Tabla 9.4. Subclaves candidatas para texto plano: 0x0x714e9a97477203fe, clave: 0x192ca36f47ea3d10, error: 0xeaa5160b.

$Key_{16}(hex)$							
$K_{16,1}$	$K_{16,2}$	$K_{16,3}$	$K_{16,4}$	$K_{16,5}$	$K_{16,6}$	$K_{16,7}$	$K_{16,8}$
04	23	2d	01	2d	0d	0b	30

Tabla 9.5. Coincidencias parciales entre las tablas 9.3 y 9.4.

Una vez se ha obtenido la clave de la última ronda a través de las subclaves parciales de cada caja de sustitución hay que deshacer las permutaciones a las que se somete a la clave en cada ronda. Dichas permutaciones consisten simplemente en una rotación en función de la ronda por lo que se puede deshacer sin problemas. Sin embargo a los 48 bits de clave hay que añadirles otros 8 bits que simplemente no se usan en cada ronda en comparación con la ronda previa. Esos 8 bits se asignarán a la clave obtenida pero sin asignarles un valor predefinido. De esta forma se desharán las permutaciones de las claves hasta llegar al principio del algoritmo cuya permutación inicial también se deshará. Una vez llegado a este paso, los 8 bits desconocidos se obtendrán mediante una batería de simulaciones de un máximo de $2^{8\,\text{bits}} = 256$ simulaciones, comprobando una a una las distintas salidas hasta que coincida con una salida de prueba.

La instrumentalización del código ha permitido comprobar la viabilidad del proceso de obtención de la clave completa desconociendo a priori el número necesario de inyecciones. Para ello se ha optado por hacer uso de un generador de textos planos que unido a una alternancia de

un mínimo de dos errores distintos consecutivos permitirá obtener la clave. Dicho generador de textos planos permitirá obtener distintos pares (C^{gold} , C^{faulty}) pero por sí sólo no reducirá la lista de subclaves candidatas por lo que será necesario hacer uso de distintos errores consecutivos, pudiendo optar por un mínimo de dos errores. Lo importante de este último paso es que el error inyectado sea distinto del anterior y dado que el texto plano es diferente en cada ataque no importa que el error se repita. Esta posibilidad de optar por dos o más errores hace que varíe el número de inyecciones.

Para reforzar el estudio del ataque se ha optado por usar un generador de bits de error pseudoaleatorios. De esta forma usando el modelo instrumentalizado del sistema completo es posible repetir el proceso de obtención de la clave un número determinado de veces que en este caso ha sido 1000 veces. Durante los 1000 procesos se ha podido obtener la clave con un mínimo de 2 y un máximo de 7 inyecciones en muy pocas ocasiones resultando en un total de 3611 inyecciones. De este modo la media es de 3,6 inyecciones por proceso de obtención de clave lo que lo sitúa entre 3 y 4 inyecciones.

7. Reproducción del ataque de fallos en FPGA

Anteriormente se ha visto la viabilidad de abordar un ataque de fallos sobre el algoritmo DES. Ahora este apartado pretende acercar más al lector a introducirse en la piel de un atacante que de verdad pretenda vulnerar el sistema. Para ello se necesitará de una implementación física del algoritmo, tal como una FPGA que se usará como soporte físico del a mismo, una herramienta para poder inyectar fallos y un script para un posterior post-procesado.

7.1 Reproducción usando la herramienta FT-UNSHADES2

Para llevar a cabo el ataque de forma real, se necesitará:

1. **Código VHDL** [33]**:** el código que implementa el algoritmo DES. Dicho código ha servido para hacer pruebas de simulación con fallos virtuales y ahora se usará como objetivo de los fallos.

2. **Herramienta FT-UNSHADES2** (Fault Tolerance University of Sevilla Hardware Debugging System) [34]**:** esta herramienta, mostrada en la figura 13 y disponible en el Departamento de Electrónica de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla consiste en un sistema compuesto de 5 FPGAs, 2 de ellas para el diseño del usuario. Este sistema permite simular el comportamiento de circuitos en entornos de prueba bajo condiciones de radiación cambiando el comportamiento de dicho circuito, y como el modelo de fallo que se pretende reproducir es el bit-flip, esto es, la inversión del valor almacenado en un biestable, que es perfectamente asimilable al caso del ataque de fallos a un core criptográfico en el que se corrompe el contenido de un registro. Por lo tanto en lo que se refiere a las inyecciones de fallos, esta herramienta servirá perfectamente para tal finalidad. Además se podrá comprobar la propagación del error inyectado en todo momento hasta la salida del circuito, permitiendo comparar los registros alterados con los originales, gracias a que la FPGA con error funcionará en paralelo con la otra que funcionará correctamente. Así mismo las salidas de las FPGAs aparecen reflejadas en los registros: GOLD (salida correcta ausente de fallo) y SEU (Single Event Upsets).

Según la figura 13 el hardware está compuesto por:

1. Daughterboard 1
2. Daughterboard 2
3. Target FPGA
4. Service FPGA
5. PCI Express connector
6. PCI connector for USB 2.0 module
7. Control FPGA
8. DDR2 module

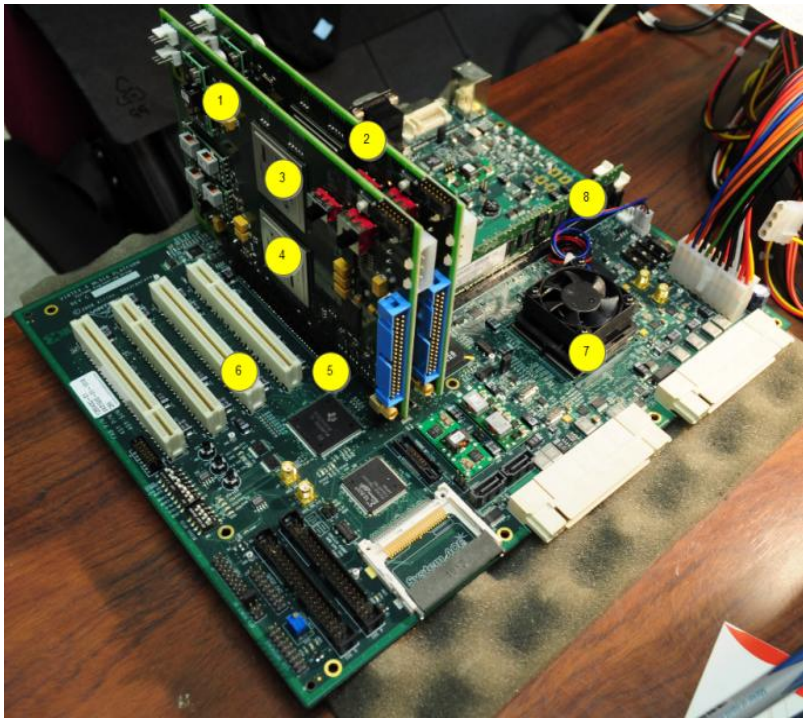


Figura 13. El hardware FT-UNSHADES2 (sin la placa de adaptación USB)

La FT-UNSHADES2 es:

- Rápida: permite una emulación acelerada gracias al uso de 2 FPGAs que funcionan en paralelo.
- Fiable: ofrece una emulación más precisa, ya que se realiza sobre una implementación del circuito (durante los procesos de síntesis e implementación se pueden producir cambios en la netlist del circuito, como por ejemplo compensaciones de Fan-out u optimizaciones de registros equivalentes, de este modo al emular el circuito en FT-UNSHADES2 se está trabajando con el resultado de dicha implementación).
- Flexible: puede inyectar cualquier error en cualquier registro y en cualquier ciclo de reloj.
- Observable: el estado interno de todos los registros puede ser observado en cualquier ciclo de reloj.
- Automatizado: el sistema puede realizar campañas de inyecciones para someter el circuito a un análisis riguroso.

- Online: permite a través de una cuenta de usuario usar el sistema de inyección de fallos para poder hacer las emulaciones que se consideren oportunas desde cualquier lugar. Esto facilita mucho su uso para cualquiera que lo necesite.

Sin embargo tiene las siguientes limitaciones:

- El diseño en VHDL viene limitado por un tamaño máximo (hasta la capacidad máxima de una xc5vfx70t que es una FPGA de alta capacidad de la familia Virtex-5).
- El sistema tiene un número máximo de pines de entrada y salida. (512 pines E/S configurables, sin contar el reloj, que va aparte).
- Cada vez que el atacante lleve a cabo un ataque con un texto plano distinto, será necesario generar los archivos necesarios para llevar a cabo la inyección. Sin embargo, esto no supone ningún problema pues como se comentó en el subapartado 6.2 el generador de textos planos es opcional. Por lo tanto no es necesario crear nuevos estímulos, siempre y cuando se inyecten distintos errores.

Las dos primeras limitaciones no son un problema para su aplicación sobre el DES. No obstante, la capacidad finita de la FPGA podría suponer una limitación para irradiar sistemas complejos como smart-cards.

Los pasos para generar los archivos necesarios para la inyección vienen explicados de forma resumida en el archivo "D8 – Final Report" del proyecto FT-Unshades2 [35]. De este modo una vez han sido generados los archivos necesarios, la propia herramienta muestra la opción para modificar el registro elegido de ser inyectado. Sin embargo para poder inyectar el fallo será necesario crear los buses asociados a dicho registro. Los buses necesarios son:

- round: bus asociado al registro que muestra la ronda de encriptación.
- inmsg: bus asociado al registro que muestra el texto de entrada en la ronda.
- outdata: bus asociado al registro que muestra el texto encriptado de salida, donde se extraerá la información de interés.

Una vez se han creado los buses que indicarán el lugar, el momento y el modo de inyectar el fallo de forma precisa, se inyectará el error elegido. Dicho error como se ha demostrado debe propagarse a través de todas las cajas de sustitución existiendo un amplio abanico de formas de hacerlo ya que basta con un error de 1 bit como mínimo en cada caja. Así mientras se cumpla dicha condición no habrá problemas.

La figura 14 muestra el funcionamiento del sistema completo basado en FPGAs. En dicha figura se puede osbservar cada una de las etapas del proceso completo pudiendo distinguir la etapa de la inyección con la herramienta FT-UNSHADES2, del post-procesado en python que se explicará en el próximo apartado.

La figura 15 presenta un ejemplo de funcionamiento de la herramienta FT-UNSHADES2 mostrando las salidas en las FPGAs, para una inyección concreta. En dicha inyección la información más importante son las salidas "Gold" y "Faulty" que se exportarán a un archivo de texto para un posterior post-procesado aparte. De esta forma dividimos el proceso de inyección de errores del de extracción de la clave en un post-procesado, asemejando un ataque real donde un atacante primero ataca (sobre el hardware real), y luego con la información recopilada (por ejemplo muestreando las salidas correctas y erróneas con un analizador lógico) intenta procesarla, utilizando dicha información para determinar bits de la clave secreta.

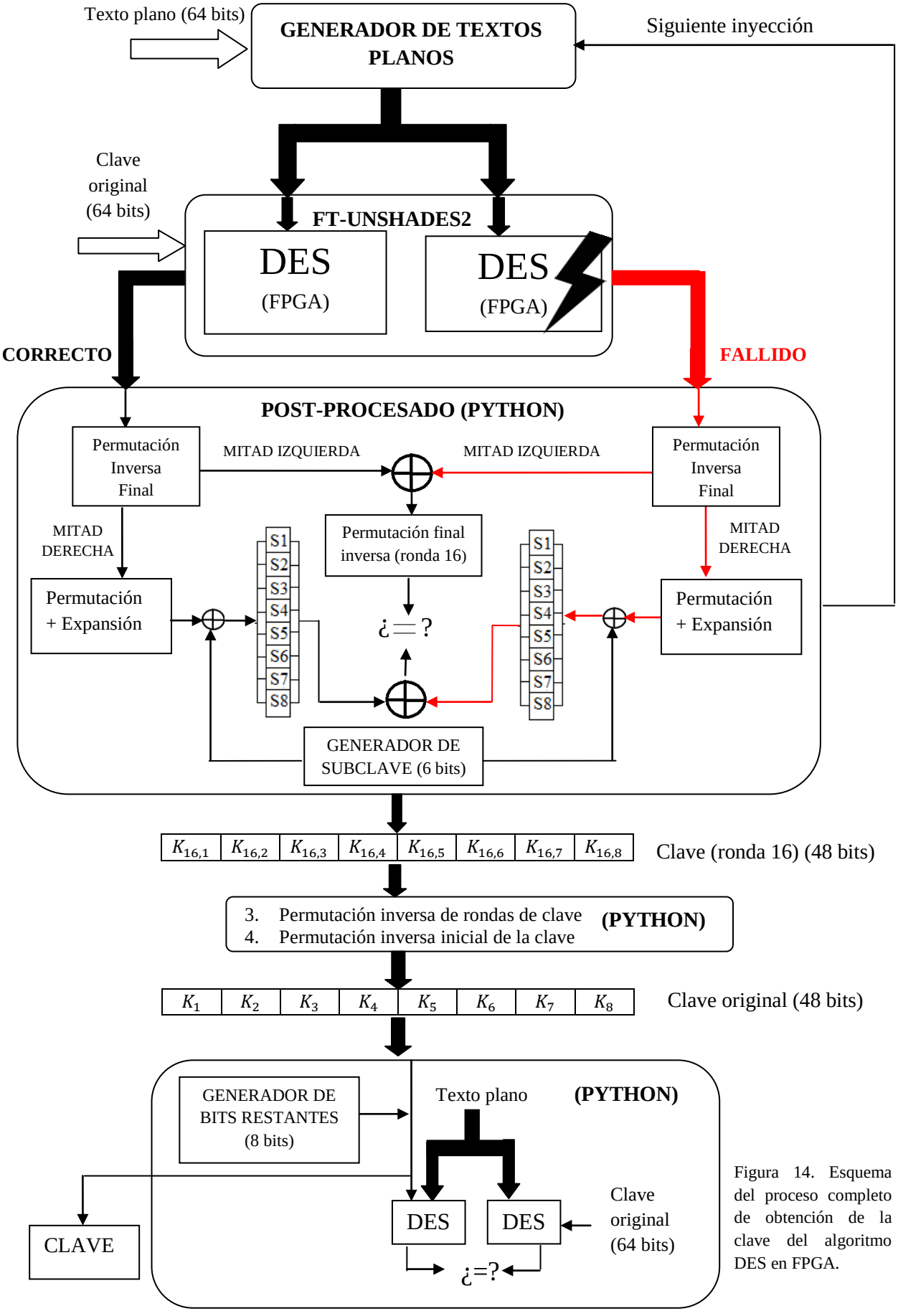


Figura 14. Esquema del proceso completo de obtención de la clave del algoritmo DES en FPGA.

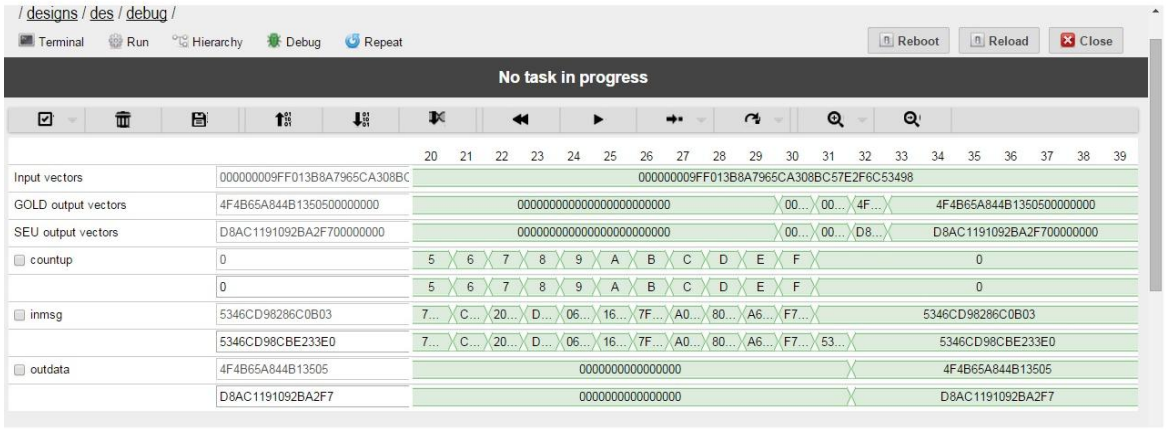


Figura 15. Ejemplo de error inyectado mediante la FT-UNSHADES2 para el texto plano: c53a69e51dc80ff9, clave: 192ca36f47ea3d10 y error: 4d9a98f2

7.2 Obtención de la clave mediante post-procesado en python.

En el subapartado anterior se ha mostrado el proceso de inyección de errores. Para ello se ha elegido un texto plano y una clave escogida al azar para luego reproducir varios fallos en la última ronda exportando la información de interés en un archivo de texto. En este archivo de texto se encuentra la información a partir del cual un script programado en cualquier lenguaje romperá el cifrado.

Para conseguir romper el cifrado encontrando la clave será necesario partir de un código que emule la encriptación DES. Pero para ello hay que elegir un lenguaje de programación sobre el que apoyarse para realizar el post-procesado, siendo python el lenguaje elegido por una serie de razones:

- Es un lenguaje de alto nivel útil para un desarrollo rápido de código.
- Su código es simple y legible.
- Posee un buen número de librerías que ayudan al desarrollo del código.
- Es un lenguaje multiparadigma donde varios estilos son compatibles (imperativo, orientado a objetos, funcional).
- Soporta bases de datos
- Es multiplataforma (Windows, MacOS y Linux), gratuito y libre.
- Aunque sea más lento de ejecutar que otros lenguajes como C o Fortran (lenguajes de bajo nivel), su simplicidad de diseño permite ahorrar tiempo ya que se adapta mejor al usuario debido a que el tiempo total de creación: $t_{total} = t_{diseño} + t_{ejecución}$, (el primer término depende del usuario mientras escribe y depura el código, y el segundo término es puramente computacional), se reduce drásticamente pues otros lenguajes de bajo nivel aunque más rápido de ejecutar requieren un tiempo de diseño mayor pues hay que dedicarle más tiempo en aprender, escribir y depurar el código, usando un direccionamiento detallado con un código repetitivo y tedioso propenso a errores.

Una vez elegido el lenguaje python, es necesario obtener un código que emule el cifrado DES [36]. El código encontrado aparte del algoritmo DES también ofrece el encriptado Triple-DES, que no se va a utilizar.

El código original posee una serie de características que le permiten encriptar cadenas de datos mayores de 64 bits de tipo Unicode con capacidad de rellenado (para cadenas no enteras de 64 bits). Estas cualidades no son necesarias en este proyecto, pues lo que se pretende es hallar la clave en formato hexadecimal a partir de textos también en formato hexadecimal. Es por esta razón por la cual el código se ha modificado con el fin de aproximar el resultado de la encriptación al visto en VHDL. Para ello se ha modificado el código de encriptación para operar directamente con tablas de datos enteros en formato hexadecimal, para un mejor acercamiento a los resultados vistos anteriormente. Además se han añadido funciones que permiten descubrir la clave en base a las mismas operaciones explicadas en apartados anteriores a partir de la lectura de los datos proporcionados por el inyector de fallos.

El script usado para obtener la clave se basa en la ejecución de los siguientes pasos:

1. Se lee en un bucle cada línea del texto exportado.
2. Para cada línea, usando C^{gold} y C^{faulty}:
    - 2.1 Se emula la ronda 16, para buscar subclaves candidatas.
    - 2.2 Si el ataque actual es el primero se vuelve al paso 1. De lo contrario se continúa en el paso 2.3
    - 2.3 Se comparan las subclaves candidatas con el fin de encontrar coincidencias parciales.
    - 2.4 Si la subclave es única se continúa en el paso 3, si no, se repite el proceso empezando en 1.
3. Se deshacen las permutaciones de la clave hasta obtener la clave del principio del algoritmo, a excepción de 8 bits desconocidos.
4. Se lleva a cabo un ataque de fuerza bruta de 8 bits hasta encontrar la clave completa.

Gracias a las modificaciones anteriores, el código presenta un comportamiento semejante al código VHDL del inicio. De este modo el atacante puede vincular el archivo de texto donde se guardan las salidas encriptadas (correctas y fallidas) procedentes del FT-UNSHADES2 siendo necesaria la creación de nuevas funciones que permitan leer el archivo donde se almacenan las parejas encriptadas, iniciando el proceso de búsqueda de subclaves candidatas. Dicho proceso repite los mismos pasos que se han visto en la instrumentalización del código, donde se ha despermutado hasta la ronda 16 para luego emularla con diversas subclaves parciales. Todo ello se ha conseguido con más facilidad en comparación con el lenguaje VHDL, pues los bucles y los procesos llamados entre sí proporcionan una mayor interactividad y simplicidad.

7.3 Resultados y experiencias.

Este apartado muestra la puesta en práctica del estudio que se ha hecho sobre el ataque en la última ronda empleando para ello la herramienta FT-UNSHADES2, la cual es usada como inyector de fallos. Además se ha empleado un script implementado en python para realizar el post-procesado en un proceso separado respecto a la inyección. De este modo se puede separar el proceso de inyección respecto al post-procesado dotando al post-procesado de mayor independencia y emulando de manera más fiel a la realidad el proceso completo de un ataque sobre el circuito real.

Para poner en práctica el ataque y tal y como se comentó en los resultados del apartado 6.3 se puede optar por usar el mismo o distintos textos planos en cada ataque. Sin embargo dado que a priori el atacante no sabe cuántas inyecciones son necesarias, se optará por usar diferentes textos planos lo que permitirá no tener una estricta condición de errores únicos, es decir, el cambiar los textos planos en cada ataque puede reducir en parte las subclaves candidatas, pero sólo hasta cierto punto a partir del cual ya sí sería necesario cambiar de error. De este modo se le dota de mayor eficacia a la inyección pues el cambio de textos planos producirá distintas parejas de textos cifrados lo que unido a diferentes errores implicará en una disminución del número de ataques.

Una vez que las diferentes inyecciones se han llevado a cabo, es hora de post-procesar las salidas del inyector, esto es, las parejas (C^{gold} , C^{faulty}) que se han obtenido en cada inyección. Dichas parejas se han exportado a un documento de texto, como refleja la figura 16.

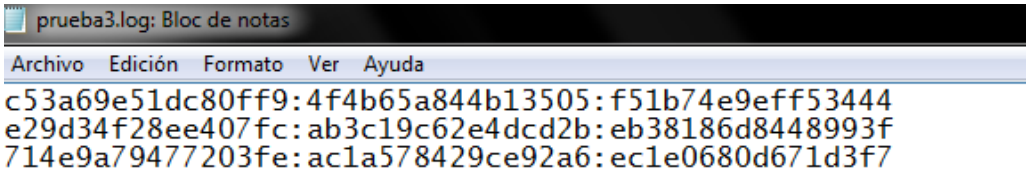
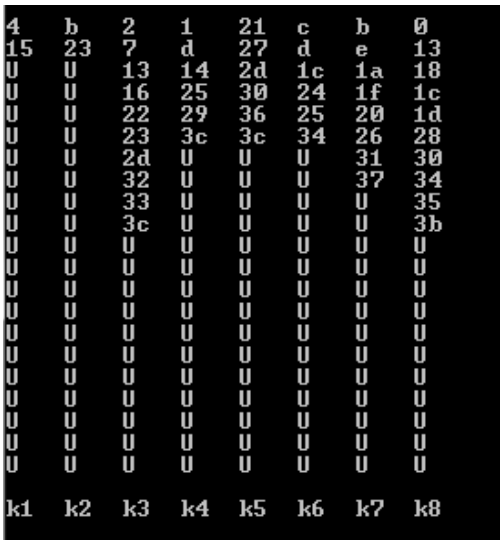


Figura 16. Texto exportado que contiene las parejas encriptadas correcta e incorrectamente en el siguiente formato: texto plano: texto encriptado correcto (C^{gold}): texto encriptado incorrecto (C^{faulty}).

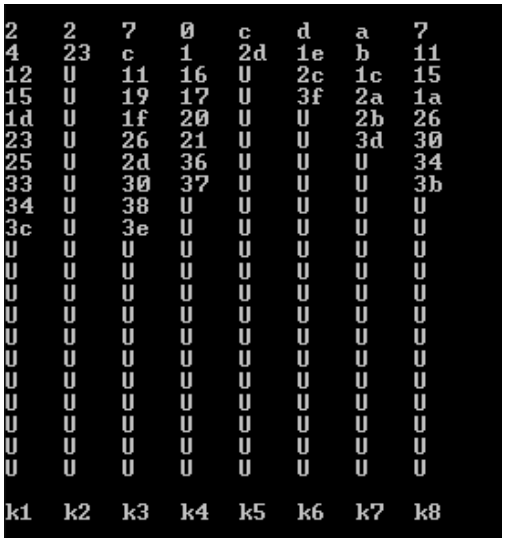
Las salidas encriptadas de dicho archivo serán usadas ahora por el script de python. Dicho script se encargará de leer las parejas (C^{gold} , C^{faulty}) sin importar la entrada que hayan tenido, ya que la información de vital importancia reside en dichas parejas, de tal forma que el proceso de completo de descubrimiento de la clave seguirá los mismos pasos que los usados en el modelo completo en VHDL. En las figuras 17.1, 17.2, 17.3, 17.4 y 17.5 se muestran los resultados de cada pareja leída y por último la clave original, la cual se ha obtenido en última instancia simulando un máximo de 256 simulaciones hasta llegar a la obtención completa de la clave original.

Como se ha podido observar la caracterización del ataque en simulación se comporta igual que en FPGA pudiendo extender el estudio llevado a cabo en simulación a un caso real.



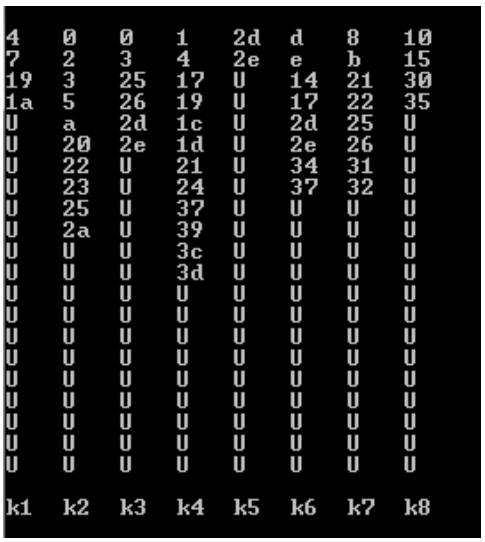
k1	k2	k3	k4	k5	k6	k7	k8
4	b	2	1	21	c	b	0
15	23	7	d	27	d	e	13
U	U	13	14	2d	1c	1a	18
U	U	16	25	30	24	1f	1c
U	U	22	29	36	25	20	1d
U	U	23	3c	3c	34	26	28
U	U	2d	U	U	U	31	30
U	U	32	U	U	U	37	34
U	U	33	U	U	U	U	35
U	U	3c	U	U	U	U	3b
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U

Figura 17.1 Primera tabla de subclaves candidatas



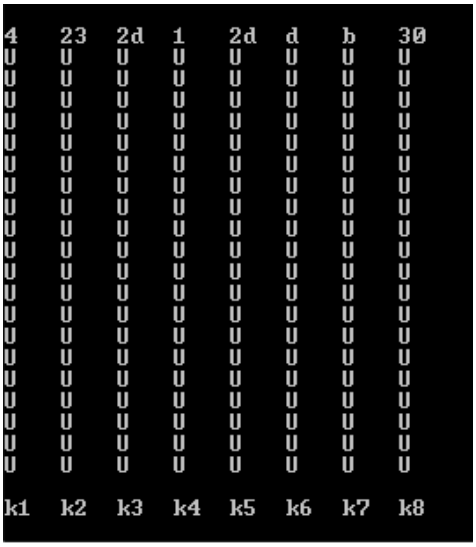
k1	k2	k3	k4	k5	k6	k7	k8
2	2	7	0	c	d	a	7
4	23	c	1	2d	1e	b	11
12	U	11	16	U	2c	1c	15
15	U	19	17	U	3f	2a	1a
1d	U	1f	20	U	U	2b	26
23	U	26	21	U	U	3d	30
25	U	2d	36	U	U	U	34
33	U	30	37	U	U	U	3b
34	U	38	U	U	U	U	U
3c	U	3e	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U

Figura 17.2 Segunda tabla de subclaves candidatas



k1	k2	k3	k4	k5	k6	k7	k8
4	0	0	1	2d	d	8	10
7	2	3	4	2e	e	b	15
19	3	25	17	U	14	21	30
1a	5	26	19	U	17	22	35
U	a	2d	1c	U	2d	25	U
U	20	2e	1d	U	2e	26	U
U	22	U	21	U	34	31	U
U	23	U	24	U	37	32	U
U	25	U	37	U	U	U	U
U	2a	U	39	U	U	U	U
U	U	U	3c	U	U	U	U
U	U	U	3d	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U

Figura 17.3 Tercera tabla de subclaves candidatas



k1	k2	k3	k4	k5	k6	k7	k8
4	23	2d	1	2d	d	b	30
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U
U	U	U	U	U	U	U	U

Figura 17.4 Resultado final del proceso de descarte de subclaves

```
CLAVE ORIGINAL:
--------------

0001100100101100101000110110111101000111111010100011110100010000
CLAVE OBTENIDA:
--------------

0001100U00101l0U1010001U0110111U0100011U1110101U0011110U00010000
```

Figura 17.5 Clave original y clave obtenida tras haber despermutado las subclaves obtenidas anteriormente y llevar a cabo un ataque de fuerza bruta de 8 bits.

8. Conclusiones y trabajos futuros

Como se ha podido observar, el proyecto ha abarcado el estudio de un algoritmo criptográfico, en este caso el DES, para poder aprovechar las vulnerabilidades propias del algoritmo y así poder romper su cifrado. Para ello se ha empleado ataque de fallos que alteran el estado interno del sistema modificando el funcionamiento normal del encriptador con el fin de poder extraer información relativa a la clave mediante un post-proccesado. Todo este proceso se ha llevado a cabo en primer lugar mediante un modelo instrumentalizado en VHDL y posteriormente mediante la herramienta FT-UNSHADES2 junto a un post-procesado programado en python. Como se ha podido observar no ha sido necesario una completa infraestructura láser.

El estudio llevado a cabo sugiere que la facilidad de llevar a cabo estos ataques no disminuirá con el decremento tecnológico del tamaño de los transistores debido a que los ataques funcionan mejor conforme más bits se alteren ya que se aumenta la probabilidad de activar más cajas de sustitución. Con la reducción de tamaños característicos de los transistores en las nuevas tecnologías, no sólo un mismo 'spot' láser puede afectar a un mayor número de registros, sino que la energía necesaria para conmutar el contenido de los mismos será menor.

El estudio de viabilidad que ha condicionado la puesta en práctica de todo el proceso en un sistema basado en FPGAs conduce a intentar mejorar este tipo de sistemas, pues el objetivo en sí del proyecto no consiste en romper un sistema de seguridad per se, sino averiguar las vulnerabilidades del mismo para poder afrontar con la mayor de las garantías un ataque que persiga fines maliciosos. De este modo ingenieros podrán testear implementaciones físicas ante fallos inyectados en la fase de diseño previa a la fabricación del dispositivo, consiguiéndose dispositivos criptográficos más robustos que ofrezcan un determinado nivel de garantía al usuario.

Algunas características del algoritmo DES permiten extraer la clave en un tiempo razonable:

- Los bits de la clave de la última ronda pueden ser obtenidos de manera independiente y concurrente ya que cada caja de sustitución funciona en paralelo.
- El proceso de deshacer las permutaciones de la clave de cada ronda no resulta difícil y además permite conseguir la clave la primera ronda.
- El ataque de fuerza bruta final resulta trivial ya que sería necesario un máximo de 256 combinaciones para hallar los bits finales.

Futuros trabajos permitirán aplicar el ataque de fallos a otros algoritmos criptográficos más complejos tales como Advanced Encryption Standard (AES) cuyo estudio está planteado en [37], y además desarrollar una metodología completa para análisis y protección de dispositivos en tiempo diseño.

8.1 Posibles protecciones

El ataque estudiado se ha podido llevar a la práctica gracias que el algoritmo es de clave simétrica ya que el uso de una sola clave tanto para encriptar como para desencriptar permite post-procesar los resultados obtenidos tras varias operaciones de cifrado.

Algunas contramedidas ya estudiadas [38] para este tipo de algoritmos se presentan a continuación:

- **Doble redundancia modular**: es el método más común de detectar errores donde diversas partes del cifrado sensibles a ataques se vuelven a ejecutar para mostrar un funcionamiento equivalente al original. Este método permite hacer comprobaciones en aquellas zonas donde se ha clonado el cifrado. Puede implementarse en paralelo o en serie:
  - Paralelo: Consiste en clonar determinados bloques para mostrar un funcionamiento equivalente al original. De este modo para pasar al siguiente paso de encriptación la contramedida chequea ambos valores obtenidos. Es muy importante que el bloque clonado no muestre ningún tipo de dependencia respecto al bloque original para que su funcionamiento sea lo más autónomo que se pueda. Un ejemplo podría ser clonar el algoritmo completo y hacer comprobaciones en pasos intermedios antes de continuar al siguiente paso. Sin embargo este método sacrifica un alto coste en hardware y consumo y además un adversario fuerte podría evitar la contramedida inyectando el mismo error en ambos bloques clonados.
  - Serie: Este método a diferencia del anterior no necesitar clonar el algoritmo ya que cuando el texto está encriptado puede volver a usar el mismo algoritmo para descifrar y obtener el texto plano de entrada para una comprobación. Esta contramedida no sacrifica hardware sino tiempo de cómputo.

- **Protección del bucle**: Consiste en proteger la integridad del contador que controla el bucle principal ya que un atacante podría modificarlo para reducir el número de rondas de encriptación y obtener la clave mediante criptoanálisis [39]. Para proteger el bucle, un contador de rondas independiente puede ser usado de diferentes formas. El segundo contador se incrementa igual que el original. Así ambos contadores deben ser siempre iguales. Otra forma de implementar el segundo contador es decrementarlo en cada ronda a partir del valor final del contador. De esta forma la suma de ambos contadores es constante.

- **Comprobación de redundancia cíclica (CRC)**: Este método permite proteger determinados registros críticos, por ejemplo el registro donde la clave está almacenada. Un atacante puede manipular dicho registro introduciendo determinados bits que alteren la salida [40]. De esta forma si el bit puesto a 1 no altera la salida significa que dicho valor de clave es correcto. Para proteger la clave se puede usar un registro basado en una suma de comprobación que asegure la integridad de la clave.

- **Cifrados involutivos**: Una operación es involutiva si coincide con su inversa (f(f(x)=x). En este tipo de cifrados, se pueden comprobar que las operaciones han sido correctas en cada uno de los pasos o rondas del algoritmo, simplemente repitiendo la misma operación sobre la salida (de ese paso o ronda) y comprobando que coincide con la entrada. Esta propiedad permite detectar errores como muestra Joshi et al. [41].

- **Modos de operación**: Los modos de operación son usados en cifrados en bloque para encriptar grandes cantidades de datos. Dichos datos deben ser divididos en bloques y el modo de operación transforma los sucesivos bloques a encriptar. El principal inconveniente de usar un modo de operación básico como ECB (Electronic codebook) es que los diferentes bloques son encriptados independientemente. Por lo tanto los datos que sean idénticos tendrán la misma salida. Para lograr una mayor dependencia entre los bloques, se suelen usar los "feedback modes". En estos modos la salida de un bloque encriptado se mezcla con el texto plano del siguiente bloque para volver a encriptar. Un ejemplo es el modo CBC (Cipher block chaining) donde la entrada de un bloque cifrado es mezclada a través de una operación XOR con el bloque encriptado previo antes de ser encriptada.
  Los "feedback modes" aseguran la confidencialidad, sin embargo, reducen el rendimiento e impiden que se paralelice la encriptación de cada bloque.

Como se ha podido comprobar este tipo de protecciones son en general aplicables a cualquier algoritmo de clave simétrica. Sin embargo tras el estudio del ataque llevado a cabo para el algoritmo DES es posible plantear un cambio del algoritmo que robustezca la encriptación. Dicho cambio reside en las transformaciones a la que se somete la clave en cada ronda de encriptación a partir de la clave original. Como se ha visto, la clave original del algoritmo DES se divide en 2 mitades a partir de las cuales se rotan uno o dos bits (en función de la ronda) y se vuelven a unir. Este proceso es fácilmente deshecho cuando la clave de la última ronda es obtenida ya que cada uno de esos procesos es independiente. Por lo tanto un planteamiento lógico sería relacionar la clave de cada ronda con la salida de la anterior ronda de encriptación (por ejemplo, a través de una operación XOR). De este modo cuando llegue la hora de despermutar la clave hasta el principio del algoritmo no será tan trivial como antes ya que ahora será necesario conocer la salida de cada ronda de encriptación.

8.2 Publicación en el congreso ICIT15

El proyecto ha concluido con la publicación de un artículo [42] en el congreso ICIT del año 2015, celebrado en Sevilla, donde se exploran de forma resumida los principales conceptos explicados en el proyecto, permitiendo así presentar ante expertos internacionales las conclusiones y trabajos futuros apoyados sobre resultados obtenidos y el estudio previo.

9. Referencias bibliográficas

[1] Colaboradores de Wikipedia. Historia de la criptografía [en línea]. Wikipedia, La enciclopedia libre, 2014 [fecha de consulta: 14 de abril del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=Historia_de_la_criptograf%C3%ADa&oldid=79134097>.

[2] Colaboradores de Wikipedia. Tarjeta inteligente [en línea]. Wikipedia, La enciclopedia libre, 2015 [fecha de consulta: 14 de abril del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=Tarjeta_inteligente&oldid=79391120>.

[3] Colaboradores de Wikipedia. Criptografía simétrica [en línea]. Wikipedia, La enciclopedia libre, 2015 [fecha de consulta: 14 de abril del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=Criptograf%C3%ADa_sim%C3%A9trica&oldid=81357096>.

[4] Estándar DES del NIST / US gov: http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

[5] Barker, William C.; Barker, Elaine (January 2012). "NIST Special Publication 800-67 Revision 1: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher". http://csrc.nist.gov/publications/nistpubs/800-67-Rev1/SP-800-67-Rev1.pdf

[6] Rivest, R. L. (1994). The RC5 Encryption Algorithm. In the Proceedings of the Second International Workshop on Fast Software Encryption (FSE) 1994, p86–96 (http://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf).

[7] AES del NIST / US gov: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[8] Schneier, Bruce: "The Blowfish Encryption Algorithm". Disponible en: https://www.schneier.com/blowfish.html

[9] Referencia IDEA: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3451

[10] Colaboradores de Wikipedia. Cifrador de flujo [en línea]. Wikipedia, La enciclopedia libre, 2015 [fecha de consulta: 14 de abril del 2015]. Disponible en <http://es.wikipedia.org/w/index.php?title=Cifrador_de_flujo&oldid=79449731>.

[11] Applied Cryptography, segunda edición, Bruce Schneier, página 397 y siguientes.

[12] Christof Paar, Jan Pelzl, "Understanding cryptography". Springer 1998.

[13] G. J. Simmons, "A survey of Information Authentication". Contemporary Cryptology, The science of information integrity, ed. GJ Simmons, IEEE Press, New York, (1992)

[14] Diffie, W. y M.E.Hellman. "New directions in cryptography", IEEE Transactions on Information Theory 22 (1976), pp. 644-654.

[15] Xin Zhou; Xiaofei Tang, "Research and implementation of RSA algorithm for encryption and decryption," Strategic Technology (IFOST), 2011 6th International Forum on , vol.2, no., pp.1118,1121, 22-24 Aug. 2011. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6021216&isnumber=6021102

[16] Referencia DSA : FIPS PUB 186-4: Digital Signature Standard (DSS), the fourth (and current) revision of the official DSA specification.

[17] Elgamal, T., "A public key cryptosystem and a signature scheme based on discrete logarithms," Information Theory, IEEE Transactions on , vol.31, no.4, pp.469,472, Jul 1985 doi:10.1109/TIT.1985.1057074

URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1057074&isnumber=22749

[18] V. Miller, "Use of elliptic curves in cryptography", CRYPTO 85, 1985.

[19] Ralph Merkle and Martin Hellman, Hiding Information and Signatures in Trapdoor Knapsacks, IEEE Trans. Information Theory, 24(5), September 1978, pp525–530.

[20] Nikita Borisov, Ian Goldberg, Eric Brewer,"Off-the-Record Communication, or, Why Not To Use PGP," https://otr.cypherpunks.ca/otr-wpes.pdf

[21] Elisabeth Oswald and François-Xavier Standaert, "Side-Channel Analysis and Its Relevance to Fault Attacks". Marc Joyce and Michael Tunstall, (Editors), *Fault Analysis in Cryptography*, Springer, 2012.

[22] Wright, P.: Spycatcher: The Candid Autobiography of a Senior Intelligence Officer. Heinemann (1987)

[23] Li, H.; Moore, S., "Security evaluation at design time against optical fault injection attacks," Information Security, IEE Proceedings , vol.153, no.1, pp.3,11, March 2006

[24] Tombs, J.; Aguirre, M.A.; Palomo, R.; Mogollon, J.M.; Guzman, H.; Napoles, J.; Rodriguez-Perez, A.; Rodriguez, J.A.; Vega-Leal, A.P.; Morillas, Y.; Garcia, J., "The experience of starting-up a radiation test at the 18MeV cyclotron in the Spanish National Accelerators Center," Radiation and Its Effects on Components and Systems, 2007. RADECS 2007. 9th European Conference on , vol., no., pp.1,5, 10-14 Sept. 2007

[25] Christophe Clavier, "Attacking Block Ciphers". Marc Joyce and Michael Tunstall, (Editors), *Fault Analysis in Cryptography*, Springer, 2012.

[26] Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: Kaliski Jr., B.S. (ed) Advances in Cryptology – CRYPTO '97. Lecture Notes in Computer Science, vol. 1294, pp. 513-525. Springer (1997)

[27] Dusart, P., Letourneux, G., Vivolo, O.: Differential fault analysis on A.E.S. In: J. Zhou, M. Yung, Y. Han (eds) Apllied Cryptography and Network Security (ACNS 2003), Lecture Notes in Computer Science, vol. 2846, pp.293-306. Springer, Berlin (2003)

[28] Giraud, C.: DFA on AES. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) Advanced Encryption Standard--AES (AES 2004), Lecture Notes in Computer Science, vol. 3373, pp. 27-41. Springer, Heidelberg (2005)

[29] Joye, M., Quisquater, J.J., Yen, S.M., Yung, M.: Observability analysis – Detecting when improved cryptosystems fail. In: B. Preneel (ed.) Topics in Cryptology--CT-RSA 2002, Lecture Notes in Computer Science, vol 2271, pp. 17-29. Springer, Heidelberg (2002)

[30] Yen, S.M., Joye, M.: Checking before output may not be enough against fault-based cryptoanalysis. IEEE Transac. Comput. 49(9), 967-970 (2000)

[31] Yen, S.M., Kim, S., Lim, S., Moon, S.J.: A countermeasure against one physical cryptoanalysis may benefit another attack. In: K. Kim (ed) Information Security and Cryptology—ICIS 2001, Lecture Notes in Computer Science, vol. 2288, pp. 269-294. Springer, Heidelberg (2002)

[32] Matthieu Rivain, "Differential Fault Analysis of DES". Marc Joyce and Michael Tunstall, (Editors), *Fault Analysis in Cryptography*, Springer, 2012.

[33] Steven R. McQueen, "Basic DES Block Cypher IP Core," http://opencores.org/project,basicdes (online)

[34] Mogollon, J.M.; Guzman-Miranda, H.; Napoles, J.; Barrientos, J.; Aguirre, M.A., "FTUNSHADES2: A novel platform for early evaluation of robustness against

SEE," Radiation and Its Effects on Components and Systems (RADECS), *2011 12th European Conference* on , vol., no., pp.169,174, 19-*23 Sept*. 2011

[35] L. Sanz, H. Guzmán, M.A. Aguirre: FT-UNSHADES2: Advanced Fault Tolerant University of Seville Hardware Debuging System, Deliverable 8 : Final Report", ESTEC Contract no. 22981/09/NL/JK

[36] Todd Whiteman, "pyDES: A pure python implementation of the DES and TRIPLE DES encryption algorithms," https://gist.github.com/eigenein/1275094 (online)

[37] Christophe Giraud, "Differential Fault Analysis of Advanced Encryption Standard". Marc Joyce and Michael Tunstall, (Editors), *Fault Analysis in Cryptography*, Springer, 2012.

[38] Jörn-Marc Schmidt and Marcel Medwed, "Countermeasures for Symmetric Key Ciphers ". Marc Joyce and Michael Tunstall, (Editors), *Fault Analysis in Cryptography*, Springer, 2012.

[39] Choukri, H., Tunstall, M.: Round reduction using faults. In: Breveglieri and Koren, I. , vol. 68, pp. 13-24

[40] Anderson, R.J., Kuhn, M.G.: Low cost attacks on tamper resistant devices. In: B. Christianson, B. Crispo, T.M.A. Lomas, M. Roe (eds) Security Protocols, *Lecture Notes in Comuter Science*, vol. 1361, pp. 125-136. Springer (1997)

[41] Joshi, N., Wu, K., Karri, R.: Concurrent error detection schemes for involution ciphers. In: Joye, M., Quisquater, J.-J., vol. 202, pp. 400-241

[42] José Manuel Martín Valencia, Hipólito Guzmán Miranda, Miguel Ángel Aguirre Echánove, "FPGA-based mimicking of cryptographic device hacking through fault injection attacks," Industrial Technology (ICIT), 2015 IEEE International Conference on.

[43] Mark Harvey, "Generating random values in VHDL testbenches," http://www.markharvey.info/vhdl/rnd/rnd.html (online)

10. Apéndice

10.1 Aproximación basada en simulación

La aproximación basada en simulación pretende estudiar la viabilidad de llevar a cabo el ataque además de poder hacer un estudio que permita conocer determinadas características con el fin de averiguar el comportamiento de los ataques con sus correspondientes consecuencias.

En la figura 18 se puede observar el esquema jerárquico que compone el sistema completo para llevar a cabo el ataque y post-procesar las salidas de manera automática. Los componentes funcionales de la simulación son:

1. TB_sistema_completo – behavior (TB_sistema_completo.vhd): componentes que estimula las entradas proporcionando el primer texto plano y la clave inicial. Además genera un registro de 32 bits aleatorios [43] usado como error cuando el sistema es usado para llevar a cabo un determinado número de procesos de obtención de clave.
   1.1 uut – sistema_completo – Behavioral (sistema_completo.vhd): controla el proceso global que inicia e interconecta los demás boques subyacentes, y además controla el bucle que asegura la repetición de pruebas de obtención de clave para errores aleatorios.
      1.1.1 My_generator_plaintext – generator_plaintext – Behavioral (generator_plaintext.vhd): generador de textos planos que partiendo de un texto plano inicial va generando diferentes textos planos en base a un registro de desplazamiento, para luego pasarlo al algoritmo DES.
      1.1.2 My_des56 – des56 – des (des56.vhd): algoritmo DES modificado con una ronda adicional donde el error será inyectado de forma específica controlando el error a inyectar o usando errores pseudoaleatorios procedentes del testbench. Esta última ronda está programada para cambiar de error conforme transcurren los diferentes ataques con sus respectivos textos planos.
      1.1.3 My_top_key – top_key – Behavioral (top_key.vhd): este bloque interconecta los diferentes componentes que se encargarán de extraer la clave mediante un post-procesado. Para ello toma del bloque anterior las salidas encriptadas correcta e incorrecta (con el fallo inyectado) para luego despermutar las últimas permutaciones sobre dichas salidas y mezclarlas entre sí mediante una transformación XOR para poder eliminar la dependencia con las rondas previas al fallo, o sea de la ronda 15 hacia atrás. El resultado obtenido resulta ser las salidas de la función de Feistel para ambas encriptaciones. Este resultado se puede convertir fácilmente en las salidas de las cajas de sustitución despermutando la permutación que sigue después de las cajas. Así que para encontrar las posibles subclaves se deberá hacer un sondeo probando con diferentes subclaves de 6 bits para cada caja de sustitución con el fin de encontrar coincidencias parciales. Para llevar a cabo este proceso será necesario emular la ronda 16 empleando para ello los mismos textos que se usaron en un principio para encriptar en el bloque anterior.

1.1.3.1 My_generator – key_generator – Behavioral (top_key.vhd): contador de 6 bits usado como generador de subclaves para cada caja de sustitución.

1.1.3.2 My_s_box1 – s_box – Behavioral(sbox.vhd): este bloque reproduce la última ronda, a excepción de la permutación final de dicha ronda, llevando a cabo una permutación expansiva sobre los 32 bits de la entrada sin fallo, convirtiéndolos en 48 bits para luego aplicar una red de sustitución cada 6 bits de los que se compone cada caja de sustitución.

1.1.3.3 My_s_box2 – s_box – Behavioral(sbox.vhd): este bloque a diferencia del anterior reproduce la última ronda pero usando los 32 bits de la entrada con el fallo inyectado.

1.1.4 My_generator_bits_inkey – generator_bits_inkey – Behavioral (generator_bits_inkey.vhd): este bloque es el encargado de generar los 8 bits restantes que se añadirán al resto de la clave inicial para llevar a cabo el ataque de fuerza bruta.

1.1.5 My_des56_basico – des56_basico – des (des56basico.vhd): este bloque implementa el algoritmo DES básico sin ningún tipo de modificación sobre el código original.

1.1.6 My_compara_salidas – compara_salidas – Behavioral (compara_salidas.vhd): comparador entre la última encriptación correcta y la salida generada en el bloque anterior.

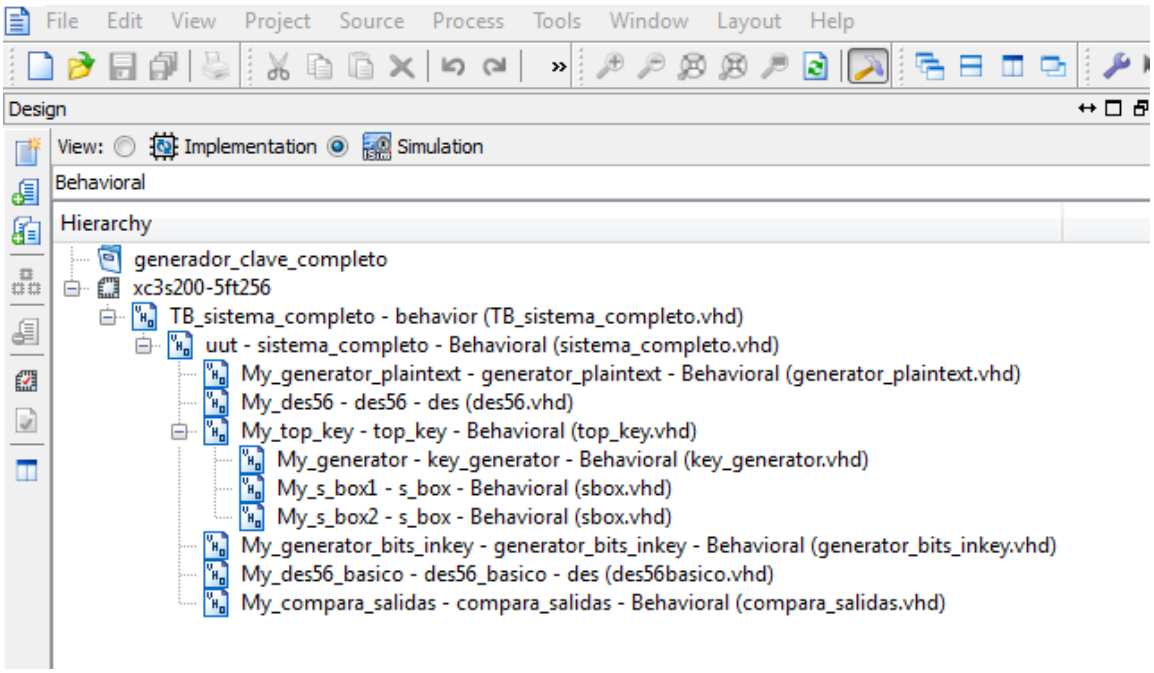


Figura 18. Componentes del sistema completo de obtención de la clave.

El ataque en simulación ha permitido comprobar la viabilidad del proceso. Sin embargo para ser más exactos y probar la veracidad de forma más general el sistema anterior tiene una doble funcionalidad mediante un generador de errores pseudoaleatorios que inyecta continuamente errores diferentes. Además se ha añadido un bucle de realimentación para poder repetir el proceso de obtención de la clave un número determinado de veces. De esta forma el sistema completo en VHDL puede extraer la clave para errores concretos y además tiene la capadidad de llevar a cabo un barrido de procesos de obtención de clave para averiguar el número medio de ataques para un total de procesos llevados a cabo.

10.2 Aproximación basada en FPGA

La aproximación basada en FPGA pretende aproximar el ataque a un escenario más real donde un atacante puede disponer de un core criptográfico DES en hardware con el fin de poder atacarlo para poder extraer la clave mediante procesado posterior.

10.2.1 Pasos para reproducir el ataque en la FT-UNSHADES2

Siguiendo los pasos descritos en el archivo "D8 – Final Report" del proyecto FT-Unshades2 [35] se generarán los distintos archivos necesarios para simular el algoritmo DES en las FPGAs. De este modo, una vez que se disponen de los archivos necesarios se siguen los pasos mostrados a continuación para inyectar un error en la ronda 16 y mostrar las salidas.

Los pasos para reproducir el ataque mediante la FT-UNSHADES2 se describen a continuación:

1. Login.
2. Design.
3. Abrir des.
4. Run this design.
5. Debug.
6. Crear buses: inmsg, outdata, countup (Figura 19).
7. jump 32.
8. writeb inmsg 5346cd98397d1a12 (cuando inmsg = 5346cd98286c0b03).
9. step 5.
10. history fpga_out 0 33 33 (C^{gold}).
11. history fpga_out 1 33 33 (C^{faulty}) .

La ejecución de los pasos aparecen en una captura de la figura 19, donde el comando: history fpga_out 0 33 33 permite mostrar la salida encriptada correcta mientras que el siguiente comando: history fpga_out 1 33 33 muestra la salida encriptada erróneamente a causa del error.

```
mkbus {countup}  {countup<3>} {countup<2>} {countup<1>} {countup<0>}


mkbus {inmsg} {inmsg<0>} {inmsg<1>} {inmsg<2>} {inmsg<3>} {inmsg<4>} {inmsg<5>} {inmsg<6>} {inmsg<7>} {inmsg<8>} {inmsg<9>} {inmsg<10>}
{inmsg<11>} {inmsg<12>} {inmsg<13>} {inmsg<14>} {inmsg<15>} {inmsg<16>} {inmsg<17>} {inmsg<18>} {inmsg<19>} {inmsg<20>} {inmsg<21>} {inmsg<22>}
{inmsg<23>} {inmsg<24>} {inmsg<25>} {inmsg<26>} {inmsg<27>} {inmsg<28>} {inmsg<29>} {inmsg<30>} {inmsg<31>} {inmsg<32>} {inmsg<33>} {inmsg<34>}
{inmsg<35>} {inmsg<36>} {inmsg<37>} {inmsg<38>} {inmsg<39>} {inmsg<40>} {inmsg<41>} {inmsg<42>} {inmsg<43>} {inmsg<44>} {inmsg<45>} {inmsg<46>}
{inmsg<47>} {inmsg<48>} {inmsg<49>} {inmsg<50>} {inmsg<51>} {inmsg<52>} {inmsg<53>} {inmsg<54>} {inmsg<55>} {inmsg<56>} {inmsg<57>} {inmsg<58>}
{inmsg<59>} {inmsg<60>} {inmsg<61>} {inmsg<62>} {inmsg<63>}



mkbus {outdata} {outdata_0} {outdata_1} {outdata_2} {outdata_3} {outdata_4} {outdata_5} {outdata_6} {outdata_7} {outdata_8} {outdata_9}
{outdata_10} {outdata_11} {outdata_12} {outdata_13} {outdata_14} {outdata_15} {outdata_16} {outdata_17} {outdata_18} {outdata_19} {outdata_20}
{outdata_21} {outdata_22} {outdata_23} {outdata_24} {outdata_25} {outdata_26} {outdata_27} {outdata_28} {outdata_29} {outdata_30} {outdata_31}
{outdata_32} {outdata_33} {outdata_34} {outdata_35} {outdata_36} {outdata_37} {outdata_38} {outdata_39} {outdata_40} {outdata_41} {outdata_42}
{outdata_43} {outdata_44} {outdata_45} {outdata_46} {outdata_47} {outdata_48} {outdata_49} {outdata_50} {outdata_51} {outdata_52} {outdata_53}
{outdata_54} {outdata_55} {outdata_56} {outdata_57} {outdata_58} {outdata_59} {outdata_60} {outdata_61} {outdata_62} {outdata_63}
precondition failed: logic map must be loaded
wait a moment...
input/des56.ll... Ok

ftu /usr/local/var/uff/jmmartin/designs/des
```

Figura 19. Captura de creación de buses para controlar la inyección del fallo en la FT-UNSHADES2.

```
history fpga_out 1 33 33
0:F51B74E9EFF5344400000000

history fpga_out 0 33 33
0:4F4B65A844B1350500000000

step 5

writeb inmsg  5346CD98397d1a12
5346CD98397d1a12

jump 32
precondition failed: device must be configured
wait a moment...
input/des56.bit... Ok
3378268 bytes sent

precondition failed: I/O vectors must be loaded
wait a moment...
input/des.dat... 41 vectors loaded

mkbus {countup}  {countup<3>} {countup<2>} {countup<1>} {countup<0>}


mkbus {inmsg} {inmsg<0>} {inmsg<1>} {inmsg<2>} {inmsg<3>} {inmsg<4>} {inmsg<5>} {inmsg<6>} {inmsg<7>} {inmsg<8>} {inmsg<9>} {inmsg<10>}
{inmsg<11>} {inmsg<12>} {inmsg<13>} {inmsg<14>} {inmsg<15>} {inmsg<16>} {inmsg<17>} {inmsg<18>} {inmsg<19>} {inmsg<20>} {inmsg<21>} {inmsg<22>}
{inmsg<23>} {inmsg<24>} {inmsg<25>} {inmsg<26>} {inmsg<27>} {inmsg<28>} {inmsg<29>} {inmsg<30>} {inmsg<31>} {inmsg<32>} {inmsg<33>} {inmsg<34>}
{inmsg<35>} {inmsg<36>} {inmsg<37>} {inmsg<38>} {inmsg<39>} {inmsg<40>} {inmsg<41>} {inmsg<42>} {inmsg<43>} {inmsg<44>} {inmsg<45>} {inmsg<46>}
{inmsg<47>} {inmsg<48>} {inmsg<49>} {inmsg<50>} {inmsg<51>} {inmsg<52>} {inmsg<53>} {inmsg<54>} {inmsg<55>} {inmsg<56>} {inmsg<57>} {inmsg<58>}
{inmsg<59>} {inmsg<60>} {inmsg<61>} {inmsg<62>} {inmsg<63>}
```

Figura 20. Captura del salto al ciclo 32 para inyectar: inmsg = 0x5346cd98397d1a12 lo que equivale a un error = 0x11111111 respecto al valor previo de inmsg=0x5346cd98286c0b03.

UNIVERSIDAD D SEVILLA

En la actualidad los algoritmos criptográficos son usados como núcleo de muchas aplicaciones computacionales. Esta tecnología aparece en un gran número de aplicaciones cotidianas como en comunicaciones seguras, almacenamiento de datos confidenciales o privados, control de acceso o smart cards usadas en bancos. Para mejorar la velocidad de cálculo y el consumo de potencia, es común que estos algoritmos estén implementados en procesadores hardware específicos para cada aplicación, por ejemplo en smart cards.

Estas implementaciones físicas de los algoritmos criptográficos son sensibles a los denominados ataques de fallos, los cuales pueden ser inducidos de diversas maneras, como por ejemplo por proyección láser sobre el chip de silicio, con el fin de cambiar el estado interno del dispositivo. De esta forma un atacante puede llevar a cabo inyecciones de errores obteniendo así parejas de textos encriptados correcta e incorrectamente. De esta forma, es posible deducir información interna del sistema, en particular los bits de la clave secreta.

El presente proyecto versa sobre la reproducción del ataque de fallos “Differential Fault Analysis” en una implementación hardware del agoritmo “Data Encryption Standard” (DES). Se reproduce el ataque tanto en simulación, para estudiar la viabilidad del mismo, como en un prototipo FPGA (Field Programmable Gate Array). Se concluye el documento aportando diversas ideas sobre cómo proteger estos circuitos a raíz de los resultados obtenidos.