Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Análisis de la seguridad de implementaciones de GIFT-COFB contra ataques por canales laterales basados en el consumo de potencia

Tesis que presenta

Rogelio Calvillo Juárez

para obtener el Grado de

Maestro en Ciencias en Computación

Directores de la Tesis

Dra. Brisbane Ovilla Martínez
Dr. Cuauhtemoc Mancillas López

Ciudad de México **Diciembre 2024**

Resumen

Este trabajo evalúa la seguridad de las implementaciones de software del cifrador GIFT-COFB frente a ataques por canales laterales, específicamente utilizando CPA (Correlation Power Analysis) para detectar posibles fugas de información. Se analizaron tres variantes del algoritmo: fixslicing, bitslicing y la versión secuencial. Los resultados indican que la variante fixslicing proporciona una mayor resistencia a los ataques por canales laterales en comparación con bitslicing y la implementación secuencial, aunque a costa de un mayor número de trazas necesarias para recuperar la clave. Fixslicing también genera más falsos positivos, lo que incrementa la dificultad para identificar correctamente la clave, pero sigue siendo posible obtenerla con un número suficiente de trazas. Por otro lado, bitslicing ofrece una menor protección, pero requiere menos trazas para ejecutar con éxito el ataque.

A pesar de que GIFT-COFB fue finalista en la competencia del NIST para algoritmos de cifrado ligero, se demostró que el modo COFB no elimina la vulnerabilidad a los ataques por canales laterales. Además, se observó que en el estado del arte existe una escasez de trabajos que exploren específicamente las implementaciones de GIFT-COFB en sus versiones fixslicing y bitslicing. Esto subraya la necesidad de realizar más investigaciones sobre la seguridad de estas implementaciones, dada su creciente relevancia en el ámbito de la criptografía ligera.

Abstract

This work evaluates the security of software implementations of the GIFT-COFB cipher against side-channel attacks, specifically using CPA (Correlation Power Analysis) to detect potential information leaks. Three variants of the algorithm were analyzed: fixslicing, bitslicing, and the sequential version. The results indicate that the fixslicing variant provides greater resistance to side-channel attacks compared to bitslicing and the sequential implementation, although at the cost of requiring a larger number of traces to recover the key. Fixslicing also generates more false positives, increasing the difficulty in correctly identifying the key, but it is still possible to recover it with a sufficient number of traces. On the other hand, bitslicing offers lower protection but requires fewer traces to successfully execute the attack.

Although GIFT-COFB was a finalist in the NIST lightweight cryptography competition, it was demonstrated that the COFB mode does not eliminate vulnerability to side-channel attacks. Additionally, it was observed that there is a lack of studies specifically exploring GIFT-COFB implementations in their fixslicing and bitslicing versions. This highlights the need for further research on the security of these implementations, given their increasing relevance in the field of lightweight cryptography.

Agradecimientos

En primer lugar dedico esta tesis a mi familia por todo su enorme apoyo durante toda mi vida.

Agradezco enormemente a mis asesores el Dr. Cuauhtemoc Mancillas López y la Dra. Brisbane Ovilla Martínez por todas sus enseñanzas y su gran paciencia.

Al igual que mis amigos de toda la vida y a mis amigos que hice del CINVESTAV, en especial a Josue Alberto con el que compartí muchas aventuras y aprendizajes.

Agradezco al Departamento de Computación del CINVESTAV Zacatenco por darme la oportunidad de estudiar su programa de posgrado, así como al CONAHCYT por brindarme la beca durante mis estudios de maestría.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Con la llegada del Internet de las cosas (IdC), estamos rodeados de dispositivos inteligentes que tiene la habilidad para comunicarse entre ellos en redes centralizadas. Estos dispositivos se utilizan ampliamente en el manejo de las cadenas de suministros, logística, casas inteligentes, control de tráfico, monitoreo médico entre otras. Aunque estos dispositivos son convenientes y eficientes para el desarrollo de tales actividades, surge el problema de la privacidad y seguridad, por lo que es necesario el uso de algoritmos criptográficos para protegerlos. El objetivo original de la criptografía es proporcionar un canal de comunicación seguro entre los diferentes actores, aunque las primitivas de criptografía modernas ofrecen servicios tales como la confidencialidad, la integridad, la autenticación de datos y entidades y el no repudio. La idea básica de estas primitivas es hacer uso de problemas matemáticos difíciles tal como la factorización de números enteros en factores primos o el logaritmo discreto en el caso de la criptografía de clave pública y la generación de secuencias pseudoaleatorias en la criptografía de clave secreta. Ambos tipos de criptografía están diseñados para que recuperar la clave secreta sea difícil computacionalmente.

Uno de los problemas principales a los que se enfrentan los dispositivos IdC es que cuentan con recursos restringidos (memoria, conjunto de instrucciones y energía de alimentación) y en muchos casos no es factible implementar una primitiva criptográfica estándar en ellos. Por la necesidad de incorporar seguridad en dispositivos IdC, el estudio de los algoritmos de criptografía ligera, o sea, algoritmos diseñados para ejecutarse en dispositivos restringidos, ha sido un área de investigación activa las últimas dos décadas. Se han propuesto muchos algoritmos para optimizar el desempeño, los recursos necesarios para su implementación y el consumo de potencia.

El criptoanálisis es una rama de la criptografía que se centra en estudiar y analizar la seguridad de los algoritmos criptográficos con el objetivo de identificar sus vulnerabilidades y puntos débiles. Los algoritmos estándar, que incluyen técnicas ampliamente utilizadas como AES (Advanced Encryption Standard) y SHA (Secure Hash Algorithm), han sido evaluados rigurosamente durante años por investigadores y expertos en criptografía. Debido a este trabajo exhaustivo, se considera que estos algoritmos son seguros contra los ataques convencionales, tales como ataques de texto plano o cifrado diferencial.

Sin embargo, en los últimos años, han surgido nuevos tipos de ataques conocidos como ataques por canal lateral (ACL, por sus siglas en inglés: *Side-Channel Attacks*). A diferencia de los métodos tradicionales, los ACL no se centran en debilidades matemáticas en el algoritmo mismo, estos explotan información derivada de la implementación física del sistema criptográfico. Desde el descubrimiento de los ACL a finales de los años noventa por Paul Kocher [4], éstos se volvieron una amenaza para los dispositivos físicos que implementan un algoritmo criptográfico. Estos ataques representan un cambio significativo en el paradigma de seguridad, ya que han demostrado que muchos algoritmos que se consideran matemáticamente seguros pueden ser vulnerables a la extracción de información secreta en un contexto físico. Los ACL explotan información física que se fuga de diversas fuentes indirectas o canales como son el consumo de potencia, radiación electromagnética o el tiempo en que un cálculo es realizado, los cuales son conocidos como canales laterales. La información contenida en las fugas obtenidas mediante la medición de los canales laterales depende de los valores intermedios calculados durante la ejecución del algoritmo criptográfico y son correlacionados con las entradas (texto plano) y la clave secreta del cifrador. Un atacante puede extraer de manera efectiva la clave secreta observando y analizando las fugas de información de los ataques laterales con instrumentos de medición y en un corto periodo de tiempo que va desde algunos minutos a algunas horas. Debido a estas razones los ACL representan una amenaza a los dispositivos IdC ya que un atacante puede tener acceso a ellos.

El GIFT-COFB es un algoritmo de cifrado simétrico de bloques diseñado específicamente para aplicaciones de criptografía ligera, en las cuales se prioriza la eficiencia y el bajo consumo de recursos. El algoritmo está pensado para implementarse en dispositivos con recursos limitados, como sensores, tarjetas inteligentes y otros sistemas embebidos. Este algoritmo fue presentado en el concurso de estandarización de criptografía ligera (*Lightweight Cryptography Standardization*) organizado por el Instituto Nacional de Estándares y Tecnología (NIST, por sus siglas en inglés). Como parte del proceso de selección, GIFT-COFB ha sido sometido a numerosas pruebas y análisis de seguridad, tanto en términos de resistencia a ataques. Los resultados indican que GIFT-COFB ofrece un balance óptimo entre seguridad y eficiencia, lo que lo hace una opción prometedora para su adopción en entornos de criptografía ligera.

Los distintos tipos de ataques por canal lateral incluyen análisis de tiempo, análisis de potencia, análisis electromagnético y ataques de inducción de fallos. Los cuales se detallan a continuación [5]:

- Ataque de tiempo: Este ataque se basa en las variaciones en el tiempo de ejecución de un dispositivo criptográfico. El atacante analiza el tiempo que tarda en procesar diferentes mensajes para deducir parámetros secretos del sistema.

- Ataque de análisis de potencia: Este ataque aprovecha las variaciones en el consumo de energía de un dispositivo durante una operación criptográfica. Existen diferentes variantes:

  - Ataque de potencia simple (SPA, por sus siglas en inglés): El ataque más básico, que consiste en analizar los trazos de potencia para identificar patrones que puedan revelar claves secretas.
  - Análisis de potencia diferencial (DPA, por sus siglas en inglés): Un ataque más avanzado que utiliza correlaciones estadísticas entre el consumo de potencia y los datos de entrada para extraer información confidencial sin necesidad de conocer la implementación interna del sistema.
  - Análisis de potencia diferencial de orden superior (HODPA, por sus siglas en inglés): Combina DPA con análisis de tiempo y criptoanálisis tradicional para aumentar la eficacia del ataque.
  - Análisis de potencia correlacional (CPA, por sus siglas en inglés): Utiliza el modelo de peso Hamming para correlacionar el consumo de energía con la distancia de Hamming, ayudando a identificar la clave correcta.
  - Ataque de plantilla: La forma más avanzada de ataque de potencia, que requiere acceso a un dispositivo idéntico para crear plantillas precisas y utilizarlas para descubrir la clave secreta.

- Ataque electromagnético: Este tipo de ataque explota las emisiones electromagnéticas de un dispositivo para obtener información sobre sus operaciones. Similar al análisis de potencia, estas emisiones pueden analizarse para extraer secretos, y debido a su capacidad para proporcionar más información, estos ataques son especialmente poderosos.

- Ataque de inducción de fallos: Los ataques de inducción de fallos alteran el funcionamiento de un dispositivo criptográfico, induciendo errores en sus operaciones para que revelen información confidencial. Los fallos pueden ser permanentes (dañando permanentemente componentes como la memoria) o transitorios (causados por alteraciones en el reloj o el voltaje).

- Ataques ópticos y análisis de tráfico:

  - *Ataques ópticos*: Explotan las emisiones de luz, como los LED, para extraer datos de dispositivos. En dispositivos con pantallas, el análisis de la intensidad de la luz puede revelar información sensible.
  - Análisis de tráfico: Estos ataques analizan los flujos de tráfico en redes de sensores para obtener información sobre la topología de la red, como la ubicación de nodos críticos, aprovechando las restricciones de energía de los dispositivos.

- Ataques acústicos y de imágenes térmicas:

  - *Ataques acústicos*: Explotan las emisiones acústicas producidas por dispositivos, como teclados o componentes de computación, para identificar teclas presionadas o procesos en ejecución.

- *Ataques de imágenes térmicas*: Utilizan imágenes térmicas para detectar la radiación infrarroja emitida por componentes, como las CPU, y revelar información sobre las operaciones internas.

A lo largo de los años, los ACL han demostrado ser extremadamente efectivos contra diversas implementaciones criptográficas. Estos ataques han sido aplicados con éxito a una amplia variedad de algoritmos, tanto de clave simétrica como de clave pública. A continuación, se presentan algunos ejemplos de ataques exitosos que han revelado vulnerabilidades en sistemas criptográficos ampliamente utilizados:

- Kocher [6] en 1996, introduce el ataque de tiempo y lo implementa a la exponenciación modular en el algoritmo RSA.
- Dhem et al. [7] utilizaron un ataque de tiempo contra una implementación real de RSA en una tarjeta inteligente.
- Schindler [8] presentó ataques de tiempo sobre la implementación de la exponenciación de RSA utilizando el Teorema Chino del Residuo.
- Hevia et al. [9] describieron ataques de tiempo sobre el algoritmo DES, específicamente para recuperar el peso Hamming de la clave secreta.
- Brumley y Boneh [10] demostraron que los ataques de tiempo podían revelar claves privadas de RSA en un servidor web basado en OpenSSL, al explotarse a través de una red local.
- Biham y Shamir [11] presentaron un análisis de fallos sobre el esquema de cifrado simétrico DES, mostrando cómo los fallos pueden ser utilizados para extraer información confidencial.
- Anderson y Kuhn [12] discutieron formas realistas de inducir fallos transitorios (*glitches*), los cuales pueden comprometer la seguridad de los algoritmos criptográficos.
- Skorobogatov y Anderson [13] propusieron un ataque óptico de fallos, demostrando que con equipos relativamente baratos se podían inducir fallos en una tarjeta inteligente al iluminar transistores específicos, revelando claves privadas de RSA.

Los ataques de análisis de potencia han demostrado ser altamente efectivos en la mayoría de las implementaciones simples de cifrados simétricos y de clave pública, como se muestra a continuación:

- Sommer et al. [14] atacaron una implementación de DES de una tarjeta inteligente, utilizando el método de análisis de potencia simple.
- Novak et al. [15] aplicó el método de análisis de potencia simple a una implementación de RSA.

- Walter et al. [16] presentaron un ataque a RSA, utilizando el método de análisis diferencial de potencia.
- Nuradha et al. [17] utilizaron el método de análisis de potencia por correlación para recuperar la clave de cifrado de AES en un microcontrolador.
- O'Fllyn et al. [18] atacaron a AES-256 ara recuperar la clave completa de 32 bytes y el vector de inicialización mediante análisis de potencia por correlación.
- Taha et al. [19] atacaron al algoritmo Keccak específicamente en su uso como MAC en un procesador Microblaze.
- Koziel et al. [20] utilizaron ataques de análisis de potencia refinados que explotan valores cero para extraer bits de la clave secreta para recuperar las claves SIDH (Supersingular Isogeny Diffie-Hellman).

En la actualidad, herramientas como ChipWhisperer [1] han facilitado significativamente el análisis de potencia en criptosistemas. ChipWhisperer es una plataforma de código abierto diseñada específicamente para realizar ataques de canal lateral. Combina herramientas de hardware y software que permiten la captura y análisis de señales eléctricas, lo que posibilita ataques como el análisis de potencia diferencial y ataques de inyección de fallos. Esta plataforma es ampliamente utilizada para evaluar la seguridad de implementaciones criptográficas en dispositivos de hardware y software, que permite explorar vulnerabilidades en sus diseños físicos.

1.1. Propuesta

La idea básica del análisis de potencia es revelar la clave secreta de un dispositivo criptográfico a través de su consumo de potencia. Esencialmente dos dependencias del consumo de potencia son explotadas: la dependencia de datos y la dependencia de operación. El consumo de potencia instantánea del dispositivo depende de los datos que procesa y de las operaciones que realiza.

Se propone realizar ataques por canales laterales a una implementación sin protección y propuestas protegidas en software de GIFT-COFB para encontrar fugas de información y analizar la vulnerabilidad de ellas a ataques por canales laterales basados en el consumo de potencia.

El enfoque que se quiere utilizar es realizar un ataque de análisis de potencia para poder extraer la clave secreta. Con la ayuda de la herramienta ChipWhisperer se planea programar un microcontrolador que será la víctima, para que realice operaciones criptográficas con el algoritmo GIFT-COFB. El microcontrolador recibirá texto plano desde la computadora, lo cifrará y enviará el resultado de regreso a la computadora. Durante el tiempo en que el cifrado sea realizado, el consumo de potencia será medido con el ChipWhisperer y registrado para realizar el análisis. Estos registros serán las trazas de las fugas de información que se utilizarán para recuperar la clave secreta.

[1] https://www.newae.com/chipwhisperer

1.2. Objetivos generales y específicos del proyecto

General

Verificar si existen fugas de información en implementaciones de los algoritmos de criptografía ligera GIFT y GIFT-COFB.

Particulares

1. Proponer modelos de ataques por canales laterales para buscar posibles fugas de información en implementaciones de GIFT y GIFT-COFB. Estas implementaciones están basadas en *fixslicing*, *bitslicing*.

2. Desarrollar las implementaciones en el firmware de ChipWhisperer para obtener las trazas de potencia de cada uno de los algoritmos analizados.

3. Verificar si las implementaciones antes mencionadas de GIFT y GIFT-COFB son seguras a ataques por canales laterales y la dificultad de estas para vulnerarlas.

1.3. Preliminares y notación

Sea $\{0,1\}^n$ el espacio de todas las cadenas binarias de longitud n, análogamente dichas cadenas son consideradas elementos del campo de Galois $GF(2^n)$, por lo que pueden ser representadas como polinomios de grado a los más n. Si $a, b \in \{0,1\}^n$, su adición es denotada como $a \oplus b$ y es calculada como una operación o-exclusiva a nivel de bits. El producto de define como $ab \mod q(x)$ donde $q(x)$ es un polinomio irreducible de grado n, es decir, no tiene raíces en el campo $GF(2)$. La operación *xtimes*, es la multiplicación del monomio x por a y reducida según el polinomio irreducible utilizado, misma que puede ser calculada de forma muy eficiente utilizando un corrimiento de bits y unas pocas operaciones $\oplus$.

Un cifrador por bloque es una función $E : \{0,1\}^n \times \{0,1\}^k \to \{0,1\}^n$, donde n es la longitud en bits del bloque y k es la longitud en bits de la clave. Se denota como $E_K(\cdot)$ y la función inversa como $E_k^{-1}()$, para todo $m \in \{0,1\}^n$ se cumple que $m = E_k^{-1}(E_k(m))$. Dado que un cifrador por bloques es invertible, es una permutación del espacio de cadenas de n bits, cada clave instancia una permutación distinta.

Un algoritmo de cifrado autenticado con datos asociados (CADA) es una función $\mathbf{AE} : \{0,1\}^m \times \{0,1\}^d \times \{0,1\}^k \to \{0,1\}^c \times \{0,1\}^\tau$ donde m, d, k, c y τ son las longitudes en bits del mensaje, de los datos asociados, de la clave, el mensaje cifrado y la etiqueta de autenticación respectivamente. Se denota como $C, \tau = AE_K(AD, M)$, recibe como entrada un mensaje y los datos asociados y entrega como salida el mensaje cifrado acompañado de la etiqueta de autenticación. La función de descifrado es $\mathbf{AE^{-1}} : \{0,1\}^c \times \{0,1\}^d \times \{0,1\}^\tau \times \{0,1\}^k \to \{0,1\}^m \times \{0,1\}^\perp$ donde $\perp$ indica que la verificación de la etiqueta de autenticación falló y por lo tanto no se regresa el

mensaje que se ha descifrado ya que puede haber errores en la integridad del mensaje o en la autenticación (se está usando una clave distinta a la utilizada para cifrar). La función de verificación solamente compara dos etiquetas y devuelve 1 o 0 según sean iguales o no. Los datos asociados no se cifran, pero sí deben ser autenticados.

1.4. Organización de la Tesis

Capítulo 2: GIFT-COFB. Se presentan las versiones *bitslicing* y *fixslicing* del algoritmo GIFT-COFB, describiendo sus características y diferencias clave.

Capítulo 3: Ataques por canales laterales. Se explica el ataque CPA, el concepto de SNR y el uso de ChipWhisperer para realizar ataques por canal lateral.

Capítulo 4: Implementación de los ataques. Se detallan los ataques realizados a las implementaciones secuencial, *bitslicing* y *fixslicing* de GIFT-COFB.

Capítulo 5: Resultados. Se presentan los resultados de los ataques, incluyendo el número de trazas y la tasa de éxito de cada implementación.

Capítulo 6: Conclusiones. Se resumen los hallazgos principales y se propone como se podría mejorar la seguridad de las implementaciones de GIFT-COFB.

Capítulo 2

GIFT-COFB

Las limitaciones en recursos de los dispositivos IdC hace que sea imposible de utilizar algoritmos de cifrados complejos como el AES en algún modo de operación, debido a que la cantidad de recursos para su implementación puede no estar disponible en estos dispositivos o ser muy limitado. Por las necesidades de seguridad y desempeño de los IdC los algoritmos de criptografía ligera[1] han sido un área de investigación activa las últimas dos décadas. Un gran número de algoritmos innovadores de cifrado han sido propuestos con el fin de optimizar varios de los criterios de desempeño y seguridad, como lo son GIFT-COFB o ASCON.

En este capítulo se explicará de forma detallada el cifrador por bloques GIFT y el modo de operación COFB.

GIFT es una Red de Sustituciones y Permutaciones (RSP), la cual es una forma de diseñar un cifrador por bloques iterado, esto significa que una cierta secuencia de cálculos que forman una ronda es repetida un número especificado de veces. Una ronda es definida como una composición de transformaciones (sustituciones y permutaciones) aplicadas a los datos de entrada, de tal manera tal que se logra cumplir el principio de Shannon de confusión y difusión. Otro cifrador basado en una RSP es el AES, el cual fue estandarizado por el NIST en el año 2001. AES no es adecuado para dispositivos con recursos restringidos debido a sus características de diseño que no estaban orientadas a dichas plataformas. Además de GIFT existen otros cifradores ligeros, algunos ejemplos son: Midori [21], Skinny [22], Simon y Speck [23].

GIFT [24] es una familia de cifradores por bloque ligeros con dos miembros: GIFT-64 y GITF-128. Los cuales tienen un tamaño de bloque de entrada de 64 y 128 bits, ambos reciben una clave secreta de 128 bits. Consisten de una función de ronda compuesta de cuatro transformaciones, la cual es iterada 28 veces para GIFT-64 y 40 veces para GITF-128. La figura 2.1 muestra 2 rondas de GIFT-64 [1].

Los datos de entrada se representan en forma de un arreglo unidimensional de *nibbles*, es decir, datos de cuatro bits. En la figura 2.1 se muestran dos rondas del cifrador GIFT. A continuación, se explicará la función de ronda.

[1]El término se refiere a algoritmos de criptografía que pueden ser implementados en dispositivos muy restringidos en recursos como memoria, energía e incluso instrucciones.

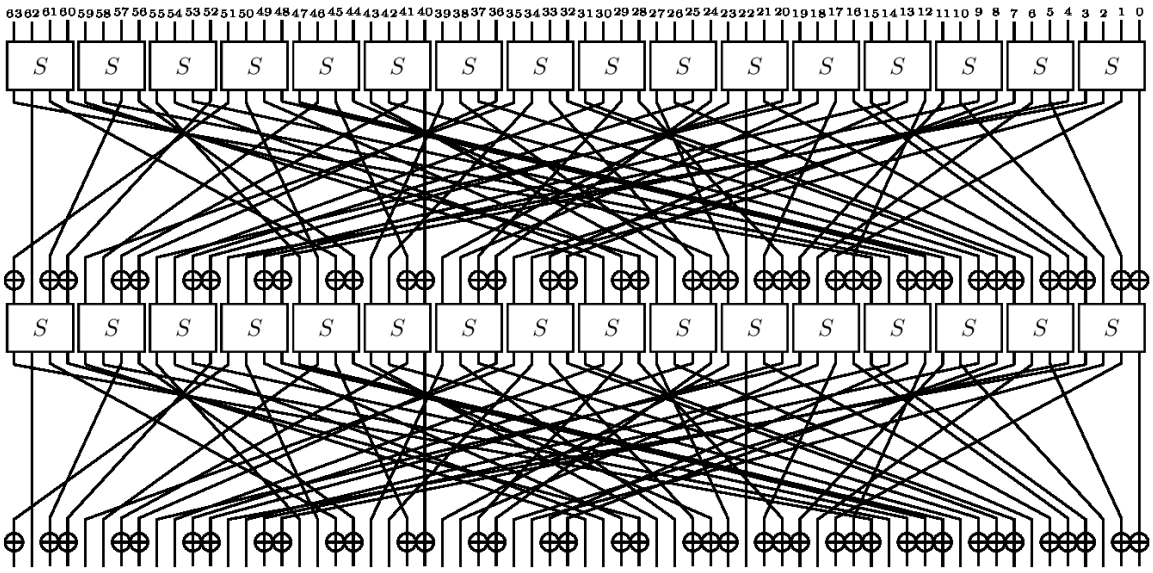


Figura 2.1: Dos rondas de GIFT-64 [1].

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	10	4	12	6	15	3	9	2	13	11	7	5	0	8	14

Tabla 2.1: Caja S.

2.0.1. Función de ronda

La función de ronda de GIFT se compone de tres transformaciones caja de substitución, permutación de bits y suma de la clave de ronda[2], las cuales se describen a continuación.

Substitución de bits. Consiste en la sustitución de cada *nibble* en el estado por un valor extraído de una tabla llamada caja S. Ambas versiones de GIFT usan la misma caja S de cuatro bits. Se representa de la siguiente forma:

$$w_i \leftarrow GS(w_i), , \forall_i \in 0, ..., s-1,$$

donde $s = 16$ para GIFT-64 y $s = 32$ para GIFT-128. Dado que S es una función del tipo $\mathcal{S} : 0,1^4 \rightarrow 0,1^4$, puede ser representada por medio de ecuaciones booleanas (ecuaciones (2.1)) o mediante una tabla de consulta (tabla 2.1).

$$\begin{aligned} x[1] &= \neg(x[1] \oplus \neg(x[0] \cdot x[2]) \\ x[0] &= \neg(x[0] \oplus \neg(x[1] \cdot x[3]) \\ x[2] &= \neg(x[2] \oplus \neg(x[0] + x[1])) \\ x[3] &= \neg(x[3] \oplus x[2]) \\ x[1] &= \neg(x[1] \oplus x[3]) \\ x[2] &= \neg(x[2] \oplus \neg(x[0]) \cdot x[1]) \end{aligned} \tag{2.1}$$

Rondas	Constantes
1-16	01,03,07,0F,1F,3E,3D,3B,37,2F,1E,3C,39,33,27,0E
17-32	1D,3A,35,2B,16,2C,18,30,21,02,05,0B,17,2E,1C,38
33-48	31,23,06,0D,1B,36,2D,1A,34,29,12,24,08,11,22,04

Tabla 2.2: Constantes de ronda.

Permutación de bits. Consiste realizar un reordenamiento de los bits en el estado conforme las siguientes ecuaciones dependiendo si se instancia GIFT-64 o GIFT-128:

$$b_{P(i)} \leftarrow b_i, , \forall_i \in 0, ..., n-1$$

$$P_{64}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 16\left(\left(3\left\lfloor \frac{i \mod 16}{4} \right\rfloor + (i \mod 4)\right) \mod 4\right) + (i \mod 4)\,,$$

$$P_{128}(i) = 4\left\lfloor \frac{i}{16} \right\rfloor + 32\left(\left(3\left\lfloor \frac{i \mod 16}{4} \right\rfloor + (i \mod 4)\right) \mod 4\right) + (i \mod 4)\,.$$

Suma de la clave de ronda. Este paso consiste en agregar la clave de ronda y las constantes de ronda. Una clave de ronda RK es obtenida del estado de la clave y es particionada en dos palabras de s bits, donde $RK = U||V = u_{s-1}...u_0||v_{s-1}||v_0$ y $s =$ 16 ó 32 respectivamente.
Suma de la clave de ronda para GIFT-64.

$$b_{4i+1} \leftarrow b_{4i+1} \oplus u_i, , b_{4i} \oplus v_i, , \forall_i \in 0, ..., 15$$

Suma de la clave de ronda para GIFT-128

$$b_{4i+2} \leftarrow b_{4i+2} \oplus u_i, b_{4i+1} \oplus v_i, , \forall_i \in 0, ..., 31$$

Para ambas versiones de GIFT, se realiza una operación $\oplus$ con un único bit 1 y una constante de ronda de seis bits $C = c_5c_4c_3c_2c_1c_0$ con el estado en las posiciones de los bits n - 1, 23, 19, 11, 7 y 3 respectivamente. Dicha constante de ronda se obtiene mediante el siguiente registro de corrimiento lineal retroalimentado (*LFSR* por sus siglas en inglés):

$$c_5||c_4||c_3||c_2||c_1||c_0 \leftarrow c_4||c_3||c_2||c_1||c_0||c_5 \oplus c_4 \oplus 1,$$

los seis bits son inicializados en cero. Las constates de ronda generadas son mostradas en la tabla 2.2.

2.0.2. Generación de claves de ronda

En ambas versiones de GIFT se utiliza una clave de 128 bits. La clave inicial es almacenada en el estado de clave k, el cual se divide en ocho palabras k_i y $|k_i| = 16$ para $i \in 0, 7$. Para generar una clave de ronda, el estado de clave se actualiza como:

$$k_7||k_6||k_5||k_4||k_3||k_2||k_1||k_0 \leftarrow k_1 \ggg 2||k_0 \ggg 12||...||k_2,$$

donde $\ggg$ representa un corrimiento circular a la derecha.

Para GIFT-64 se extrae una clave de ronda de 32 bits como $RK = U||V$,

$$U = k_1, V = k_0.$$

En el caso de GIFT-128 la clave es de 32 bits,

$$U = k_5||k_4, V = k_1||k_0.$$

RK es utilizada en la suma de la clave de ronda.

GIFT-COFB

COFB (*Combined FeedBack*, por sus siglas en inglés) es un esquema de cifrado autenticado con datos asociados. COFB recibe bloques de mensaje o datos asociados de 128 bits, una clave secreta de 128 bits y un *nonce*[2] del mismo tamaño. Utiliza como bloque básico un cifrador por bloques, adicionalmente utiliza una función lineal compuesta de $\oplus$ y multiplicaciones en un campo finito $GF(2^{64})$ con diferentes constantes pequeñas como 2 y 4 [25]. En la figura 2.2 se muestra cómo COFB opera cuando recibe bloques de mensaje y de datos asociados.

COFB se enfoca principalmente fue diseñado como un algoritmo ligero y que lograr utilizar una sola llamada al cifrador utilizado por cada bloque de entrada. Una característica importante de este modo de operación es que no necesita que el cifrador subyacente sea invertible durante el descifrado, lo que permite un tamaño del estado interno pequeño.

Descripción de los bloques básicos de construcción

Clave y bloque del cifrador: la primitiva criptográfica subyacente es un cifrador E_K con bloques de n-bits. Se asume que n es múltiplo de 4, ϵ denota una cadena de longitud cero y K es la clave del cifrador por bloques.
Función de relleno. Para $x \in \{0,1\}^*$, se define la función de relleno como:

$$Pad(x) = \begin{cases} x & \text{si x} \neq \epsilon \text{ y } |x| \bmod n = 0 \\ \text{de lo contrario.} & x||10^{(n-(|x|;mod;n)-1)} \end{cases}$$

[2]Se refiere a un valor que debe ser distinto para cada mensaje/datos asociados que sean procesados con la misma clave.

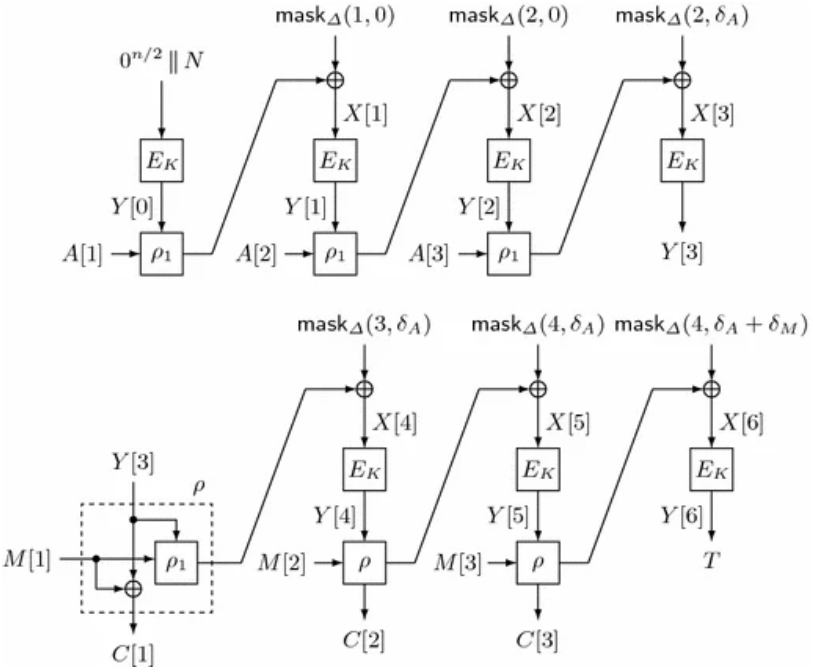


Figura 2.2: Cifrado autenticado de COFB para tres bloques de datos asociados y texto plano.

Función de enmascaramiento. Es una función $Mask : \{0,1\}^{n/2} \times \mathbb{N} \times \mathbb{N} \to \{0,1\}^{n/2}$ definida como:

$$mask(\Delta, a, b) = \alpha^a \cdot (1 + \alpha) \cdot \Delta.$$

Se puede escribir $mask(\Delta, a, b)$ como $mask_\Delta(a, b)$. Aquí, $\cdot$ se entiende como la multiplicación sobre $GF(2^{n/2})$ y α como el elemento primitivo del campo; el polinomio irreducible que define el campo es $p(x) = x^{64} + x^4 + x^3 + x + 1$.

Función de retroalimentación. Sea $Y \in \{0,1\}^n$ y $Y[1], Y[2] \xleftarrow{n/2} Y$, donde $Y[i] \in 0, 1^{n/2}$. Se define una función $G : \{0,1\}^n \to \{0,1\}^n$ como:

$$G(Y) = (Y[2], Y[1] \lll 1)$$

donde $X \lll r$ es la rotación circular a la izquierda de la cadena X en r posiciones. Ahora se define $\rho_1(Y, M) = G(Y) \oplus M$, y finalmente la función de retroalimentación ρ y su inversa ρ^{-1} se definen como:

$$\rho(Y, M) = (\rho_1(Y, M), Y \oplus M),$$

y

$$\rho^{-1}(Y, c) = (\rho_1(Y, Y \oplus C), Y \oplus C).$$

Es importante notar que recientemente el NIST realizó un proceso un mecanismo

de estandarización de diferentes algoritmos de criptografía ligera del NIST [3], donde GIFT-COFB fue uno de los finalistas.

2.1. Técnicas de implementación en arquitecturas Cortex-M

2.1.1. Bitslicing

La técnica de *bitslicing* consiste en implementar las cajas S utilizando instrucciones lógicas a nivel de bits. Debido a que estas instrucciones son independientes del mensaje de entrada y de la clave, generalmente las implementaciones con *bitslicing* son resistentes a los ataques de temporización. Su objetivo es reducir el número de instrucciones de almacenamiento y carga en la memoria [26].

La representación *bitslicing* para la permutación GIFT puede verse como un rectángulo, es decir el bitslice está en dos dimensiones (2D) con el propósito de tener una implementación más eficiente. GIFT en 2D está compuesto por tres pasos: SubCells, PermBits y AddRoundKey [24].

Initialization. El texto plano es acomodado en cuatro filas de 16 ó 32 bits de arriba hacia abajo y de derecha a izquierda. El estado del cifrador es visto como una matriz de dos dimensiones.

Para utilizar esta técnica se deben de reordenar las entradas para tener una representación apta para el *bitslicing* [2]. Para ello se utiliza la siguiente función de intercambio:

$$\text{SWAPMOVE}(A, B, M, n): \ T \ = (B \oplus (A \gg n)) \wedge M \ B = B \oplus T \ A \ = A \oplus (T \ll n)$$

La función consiste en intercambiar los bits de B enmascarados con M con los bits de A enmascarados con el corrimiento $M \ll n$.

SubCells. Ambas versiones de GIFT (GIFT-64, GIFT-128) usan la misma caja S invertible de cuatro bits. La caja S es aplicada en paralelo a cada columna del estado del cifrador (en forma de matriz), todas las cajas S puede ser ejecutadas en paralelo utilizando solo 13 operaciones como se describió en la sección 2.1.

PermBits. Se aplican cuatro permutaciones a nivel de bits a las filas del estado del cifrador independientemente. Mapea los bits de la posición (i, j) a la posición $(i, P_i(j))$. Este es el paso más costoso ya que se requiere mover bit a nivel de software, para realizar una permutación P_0 de 16 bits a S_0 [2]:

[3] National Institute of Standards and Technology, https://csrc.nist.gov/projects/lightweight-cryptography

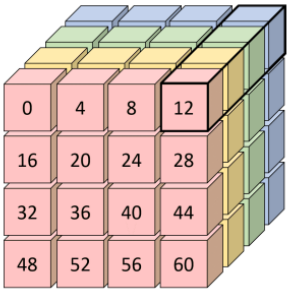


Figura 2.3: Representación cúbica del estado de GIFT-64, cada color hace referencia a un distinto slice mientras que los bits seleccionados en negro son a los cuales se les aplica la caja S [2].

$$\begin{aligned} P_0(S_0) = (S_0 \wedge \texttt{0x0401}) \vee ((S_0 \wedge \texttt{0x0002} \ll 11) \\ \vee ((S_0 \wedge \texttt{0x0020}) \ll 8) \vee ((S_0 \wedge \texttt{0x0008} \ll 1) \\ \vee ((S_0 \wedge \texttt{0x2000}) \ll 2) \vee ((S_0 \wedge \texttt{0x0040} \ll 3) \\ \vee ((S_0 \wedge \texttt{0x0200}) \ll 5) \vee ((S_0 \wedge \texttt{0x0004} \ll 6) \\ \vee ((S_0 \wedge \texttt{0x1000}) \ll 9) \vee ((S_0 \wedge \texttt{0x8000} \ll 8) \\ \vee ((S_0 \wedge \texttt{0x0100}) \ll 6) \vee ((S_0 \wedge \texttt{0x0800} \ll 5) \\ \vee ((S_0 \wedge \texttt{0x4010}) \ll 3) \vee ((S_0 \wedge \texttt{0x0080} \ll 2) \end{aligned} \tag{2.2}$$

AddRoundKey. Una fracción de $n/2$ de la clave es extraída del estado de la clave y se realiza la función `XOR` a las primeras 2 filas del estado del cifrador. Para la versión de GIFT-64.

2.1.2. Fixslicing

Esta representación tiene el objetivo de disminuir la latencia generada al realizar la permutación en la representación bitslice. Al hacer *fixslicing* se divide el estado cada cuatro bits. Cada bit se coloca en cuatro diferentes slices. Esta nueva representación puede verse gráficamente como un cubo en la figura 2.3.

SubCells. La aplicación de la subtitución con las cajas S permanece igual que en la representación bitslice. Las 16 cajas S para GIFT-64 se aplican en paralelo [2].

PermBits. Para realizar la permutación se realizan los siguientes pasos:

- Obtener la transpuesta de cada slice.
- Aplicar intercambios en las filas:
  - Slice 0: intercambio fila 1 con 3.
  - Slice 1: intercambio fila 0 con 1 y fila 2 con 3

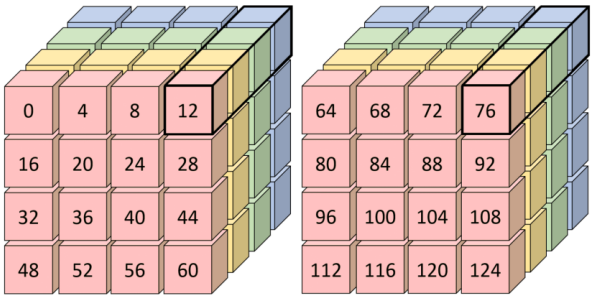


Figura 2.4: Representación cúbica del estado de GIFT-128, cada color hace referencia a un distinto slice mientras que los bits seleccionados en negro son a los cuales se les aplica la caja S [2].

- Slice 2: intercambio fila 0 con 2.
- Slice 3: intercambio fila 0 con 3 y fila 1 con 2.

AddRoundKey. Se mantiene igual que en la representación Bitslice.

Para la versión GIFT-128 se ordenan los bits en dos cubos, tal como se muestra gráficamente en la figura 2.4. Se aplican las 32 cajas S en paralelo. La permutación se define de la siguiente manera:

- Se obtiene la matriz transpuesta de cada slice de los cubos.
- Se mezclan las matrices izquierdas y derechas de cada slice.
- Se aplican los siguientes intercambios de filas:
  - Slice 0: intercambia las 2 mitades de abajo.
  - Slice 1: intercambia las mitades de abajo y arriba de los slices independientemente.
  - Slice 2: intercambia las 2 mitades de arriba.
  - Slice 3: intercambia la mitad de arriba y abajo de forma cruzada.

Capítulo 3

Ataques por canales laterales

Los ataques por canales laterales (ACL), introducidos en 1996 por Paul Kocher [4] explotan las fugas de los canales laterales, tal como el consumo de potencia de un dispositivo para extraer información secreta. Los ACL pueden ser clasificados en dos categorías: enfocados o no enfocados. Los ataques enfocados requieren acceso a un dispositivo, lo cual es una suposición necesaria que en la práctica no siempre es posible. Los ataques no perfilados incluyen el análisis de potencia diferencial (APD), el análisis de potencia correlacional (APC) y el análisis de información mutua (AIM).

3.1. Ataques de análisis de potencia

Este tipo de ataques toman ventaja de la información que puede ser obtenida al medir la potencia consumida por un dispositivo criptográfico. La mayoría de la potencia consumida por estos dispositivos es usada principalmente cuando los transistores cambian de estado. Esto es conocido como consumo dinámico y la cantidad de potencia consumida en el cambio de estado puede ser usada para poder hacer un criptoanálisis. También es conocido que el consumo de potencia estático de los dispositivos tiene relación con el último valor almacenado, el cual puede ser usado por un atacante.

Muchos dispositivos criptográficos son simples microcontroladores con un único hilo de ejecución, sin *pipeline* y con un ciclo de reloj lento. Debido a estas características es relativamente fácil encontrar una relación entre los datos e instrucciones procesadas y la potencia consumida. De esa relación, un atacante puede suponer los valores de los datos procesados. Algunos de estos valores son datos secretos almacenados en el microcontrolador.

El análisis de potencia utiliza las trazas del consumo de potencia medido durante la operación del dispositivo criptográfico. Una traza es un conjunto de mediciones del consumo de potencia obtenidas durante una operación de cifrado.

3.1.1. Análisis de potencia simple

Los ataques de potencia simple (APS) es un método que involucra una interpretación directa de las mediciones del consumo de potencia las cuales son adquiridas durante las operaciones de cifrado realizadas en un dispositivo. Diferentes valores de potencia sobre la traza son debido a que diferentes instrucciones forman parte de un algoritmo de cifrado. Debido a que la secuencia de instrucciones puede ser revelado mediante un APS y la ruta de ejecución tiene una relación directa con los datos procesados, el APS puede ser usado para romper la implementación criptográfica.

En un ataque de análisis de potencia el atacante debe tener acceso a algunas trazas de potencia y para poder tener un ataque exitoso, debe conocer a profundidad el algoritmo criptográfico que se utiliza, así como, los detalles de la implementación del mismo, para poder conocer los pasos intermedios del algoritmo.

3.1.2. Análisis de potencia diferencial

El análisis de potencia diferencial (APD) es una técnica avanzada que usa métodos estadísticos con el propósito de identificar una relación entre las medidas de potencia y un consumo de potencia supuesto del dispositivo [27]. A diferencia de un APS, en un APD no se necesita tener un conocimiento detallado del dispositivo víctima. También se puede obtener éxito al obtener la clave secreta incluso si existe ruido en las trazas de potencia medidas. Una desventaja de este ataque es que requiere una gran cantidad de trazas de potencia.

Con el fin de obtener la clave secreta, un atacante debe tener acceso al dispositivo víctima y proponer un modelo de consumo de potencia, también se asume que se conoce el algoritmo criptográfico que la víctima está ejecutando para poder obtener las mediciones del consumo de potencia, la implementación del algoritmo y los texto planos (o en claro).

Para realizar un APD, se necesita que muchos textos planos sean cifrados y del proceso de cifrado de cada uno de los textos, medir el consumo de potencia, almacenarlo y sincronizarlo. El objetivo del ataque es comparar todas las trazas a lo largo de la ejecución del proceso de cifrado y poder hacer una suposición del instante de tiempo en el que se están realizando las operaciones que componen el algoritmo.

Debido a que el análisis de potencia diferencial confía en una comparación estadística de múltiples trazas, contramedidas de aleatorización implican que un número aleatoria es usado en cada ejecución puede dificultar o impedir el éxito de un APD.

3.2. Distinguidores

Un distinguidor es una herramienta estadística cuyo propósito es determinar la clave secreta más probable de entre un conjunto. Varias clasificaciones de distinguidores pueden ser consideradas y por lo tanto, divididos en dos grupos de acuerdo a el número de muestras y el tipo de fuga que se quiere aprovechar para recuperar información en monovariados y multivariados. Otra forma de clasificarlos es separar los distinguidores de acuerdo a la suposición de que el atacante tiene una copia del dispositivo que se quiere atacar, estos dos grupos en la literatura especializada son los ataques enfocados o no enfocados respectivamente. De hecho, si el atacante tiene un dispositivo equivalente a la víctima, entonces se puede enfocar a una fuga específica antes de la fase del ataque. La fase de enfoque consiste en estimar la distribución de probabilidad de fuga, es decir, estimar los diferentes momentos estadísticos como la media, varianza, etc. Mientras la fase de ataque consiste en calcular la diferencia entre la distribución de la fuga actual y la distribución de fuga estimada. Estas diferencias pueden ser calculadas mediante el cuadrado de la distancia o la vecindad máxima. Una tercera forma en la que se pueden clasificar los distinguidores, se basa en la implementación que se quiere atacar, es decir, si tienen como objetivo implementaciones no protegidas, específicamente las no enmascaradas. Este tipo de distinguidores son conocidos como de primer orden. Por el otro lado, están los distinguidores que tienen como objetivo implementaciones protegidas mediante el enmascaramiento, que son los distinguidores de orden superior [28].

3.2.1. Correlacional

El análisis de potencia correlacional (APC) es un método estadístico que es empleado para deducir la clave secreta correcta usando un coeficiente de correlación. El APC fue introducido en [29], usando el coeficiente de correlación de Pearson. En un ataque APC el modelo de consumo de potencia utilizado es el peso de Hamming que relaciona el consumo de potencia de un algoritmo criptográfico que esta realizando el cifrado de textos planos en un dispositivo y este consumo es usualmente proporcional a cuantos bits han cambiado de valor en un registro especifico o posición de memoria. Este es visto como el peso de Hamming entre el valor previo y el nuevo valor. El peso de Hamming es calculado como el número de unos que hay en un arreglo de bits.

Si W denota la potencia medidad y H el peso de Hamming entre los valores supuestos y los valores intermedios D, el coeficiente de la correlación de Pearson $\rho_{W,H}$ entre W y H puede ser calculado como:

$$\rho_{W,H} = \frac{Cov(W,H)}{\sigma_W \sigma_H} = \frac{E((W-\mu_W)(H-\mu_H))}{\sigma_W \sigma_H}$$

donde μ_W y μ_H son los valores intermedios respectivos, σ_W y σ_H son las respectivas desviaciones estándar, Cov la covarianza y E la media.

El valor de la clave secreta que maximiza el valor absoluto de el coeficiente de correlación es el valor que maximiza la correlación entre el valor supuesto y el valor

medido; esto es, la clave supuesta escogida es la que esta asociada el valor absoluto del coeficiente de correlación más alto.

El APC requiere menos trazas que el APD para lanzar un ataque exitosos ya que *"todos los bits de datos no supuestos correctamente penalizan a la relación de señal a ruido"* [[29], [30]]. Debido a esto, el APC es probablemente el tipo de ataque de análisis de potencia más utilizado.

3.2.2. Información Mutua

La información mutua es un concepto de teoría de la información esta mide la cantidad de información que dos variables aleatorias (X, Y) comparten de dos espacios discretos $\mathcal{X}$ y $\mathcal{Y}$, cada uno con una densidad de probabilidad de P_x y P_y. En pocas palabras, cuantifica la información de obtener a X si se observa Y. En el procesamiento de señales ayuda a comprender la dependencia de cada una, es específico para los ACL indica la cantidad de información que puede filtrarse [31]

$$\left[I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \left(\frac{p(x,y)}{p(x)p(y)} \right) \right]$$

Donde (p(x, y)) es la distribución conjunta de (X) e (Y), (p(x)) es la distribución marginal de (X) y (p(y)) es la distribución marginal de (Y).

3.3. Modelos de consumo de potencia

Los modelos de potencia son usados para simular el consumo de potencia de un dispositivo criptográfico, destacando que entre más exacto sea, los recursos computacionales necesarios para su ejecución son mayores. Siempre existe un compromiso entre la precisión de los valores dados por las herramientas estadísticas y los recursos de cómputo utilizados. Los atacantes usan modelos de potencia abstractos de gran nivel con el fin de suponer como el consumo de potencia está relacionado a los datos procesados. Durante la fase del diseño del ataque, diferentes modelos de consumo pueden ser usados. Los modelos más comunes son:

Peso de Hamming

En el ataque implementado por Kocher [27] basado en el análisis de consumo de potencia, el modelo utilizado fue el peso de Hamming (PH). En este modelo, el atacante supone que el consumo de potencia es proporcional al número de bits cuyo valor lógico es '1' en el dato que está siendo procesado, ignorando los valores previos y siguientes. Por ejemplo, en el caso de los microprocesadores con buses precargados, el consumo de potencia puede depender del peso de Hamming de los datos en el bus.

Esto es típicamente el caso si los valores precargados son «todos ceros» lo cual genera que el consumo de potencia depende de $W_H(0...0 \oplus Y_i) = W_H(Y_i)$.

Distancia de Hamming

Cuando un dispositivo criptográfico no tiene precargados ciertos valores en su entrada, el modelo de potencia que mejor se ajusta a la potencia consumida es el modelo de la distancia de Hamming. El modelo de la distancia de Hamming considera cuántas transiciones de 0 a 1 y de 1 a 0 ocurren en un dispositivo durante un periodo de tiempo dado. El modelo de la distancia de Hamming está basado en la hipótesis en donde la transición de 0 a 1 tiene un consumo de potencia similar al de la transición de 1 a 0 y también se aplica para la transición de 0 a 0 y de 1 a 1 en donde el consumo de potencia es el mismo [32]. Por lo que es aplicable a implementaciones de hardware, específicamente a las fugas emitidas por registros y buses de datos.

Otros modelos

Algunas operaciones criptográficas requieren multiplicaciones. Se sabe que el consumo de potencia es menor si uno de los factores de la multiplicación es cero, a diferencia del caso general en donde ambos operadores son diferentes de cero. Por lo tanto algunas veces los modelos de consumo tienden a suponer un consumo nulo si cualquiera de los operandos es cero y asignan 1 si ambos son diferentes de cero.

3.4. Relación señal a ruido

Los ACL están estrechamente relacionados fenómenos físicos observables causados en la ejecución de tareas en el hardware. Por ejemplo, el tiempo de ejecución y energía consumida pueden ser utilizadas para los ACL. También se pueden utilizar las emisiones electromagnéticas, la disipación calor, incluso el ruido producido. Todas estas fuentes de información que se filtran las computadoras y procesadores al realizar tareas pueden ser explotadas por adversarios maliciosos [33]. Cuando se puede medir esta información, y la señal fluctúa dependiente de los datos de entrada, operaciones y ruido aleatorio, puede usarse como ACL [34].

Con esta señal se forma una traza de canal lateral. La traza tiene tres componentes en el caso de consumo de potencia: señal de componente V_{datos}, correlacionada con el modelo de potencia dependiente de la entrada, el componente de ruido V_{op}, sin correlación con el modelo de potencia, pero dependiente de las operaciones criptográficas y el componente de ruido aleatorio V_{ruido} [34]. Se puede resumir que hay una fuga de información dependiente de los datos **señal** y una fuga independiente del ruido **ruido**.

La relación señal ruido RSR es la correlación de la señal y el componente de ruido ambiente que hay en una medición. Se utiliza como métrica para evaluar el diseño de contramedidas contra los ACL. Con RSR se obtienen dos criterios importantes: el

coeficiente de correlación y el número de trazas necesarias para obtener la clave [34]. En un ACL, la RSR representa la proporción de la información disponible respecto a la información total cuando el objetivo ejecuta un algoritmo criptográfico. RSR mide la varianza de una señal contra la varianza del ruido. Este modelo está compuesto de tres partes:

- Obtener la señal: Medir la señal y calcular su media.
- Obtener el ruido: Se sabe que existe cierto ruido aleatorio, a las trazas se les resta la media.
- Calcular la RSR: Se obtiene la varianza de las señales y el ruido.

[RSR $=$ Var(señal)$\frac{}{Var(\text{ruido})][Var(x)=\sigma^2=\frac{1}{N}\sum_{i=1}^{N}(x_i-\bar{x})^2]Donde}\sigma^2$, Var es la varianza, N número total de elementos, x_i cada valor individual y $\bar{x}$ media aritmética.

3.5. Descripción de un ACL

El objetivo de un ACL es revelar las claves secretas de los dispositivos criptográficos basado en un gran número de trazas de potencia que han sido registradas mientras el dispositivo está cifrando y descifrando diferentes bloques de datos. En este caso se tiene acceso al dispositivo bajo ataque por lo que se puede seleccionar un conjunto de datos de entrada conocidos, supongamos el ataque a un cifrador por bloques que tiene como entradas un mensaje y una clave de 16 bytes (128 bits) y una única salida de la misma longitud. Se describe un ataque genérico que recupera $f - bits$ de la clave [35]. Primero se seleccionan D entradas conocidas y a continuación se sigue el siguiente procedimiento:

Paso 1: Elegir un resultado intermedio del algoritmo ejecutado. El primer paso de un ataque es elegir un resultado intermedio del algoritmo criptográfico que es ejecutado por el dispositivo atacado. Este resultado intermedio necesita ser una función $f(d, k)$ con salidas y entradas de $f - bits$, donde d es un valor no constante de la entrada conocida (hay D valores) y k son $f - bits$ de la clave. Los resultados intermedios que cumplan esta condición pueden ser usados para revelar k. En la mayoría de los escenarios ataques, d corresponde a es el texto plano o el texto cifrado.

Paso 2: Medir el consumo de potencia. El segundo paso del ataque es medir el consumo de potencia del dispositivo criptográfico mientras cifra o descifra D diferentes bloques de datos conocidos. Para cada una de estas operaciones, el atacante necesita saber el valor d correspondiente que esté involucrado en el cálculo de los resultados intermedios elegidos en el paso 1. Estos valores conocidos pueden ser expresados como un vector $d = (d_1, ..., d_n)'$, donde d_i denota el valor en la i^{th} operación de cifrado o descifrado. Durante cada una de estas operaciones el atacante registra las trazas de potencia. Las trazas que corresponden a los bloques d_i son el vector $t'_i = (t_{i,1}, ...t_{i,T})$,

donde T denota la longitud de la traza. [1] El atacante mide una traza para cada uno de los D bloques de datos y por lo tanto las trazas pueden ser escritas como una matriz $\mathbf{T}$ de tamaño DxT. Es importante para los ataques que las medidas estén alineadas correctamente. Esto significa que los valores del consumo de potencia de cada columna t_j de la matriz $\mathbf{T}$ necesita ser de la misma operación. La matriz T queda como:

$$\mathbf{T} = \begin{bmatrix} t_{1,1} & \cdots & t_{1,T} \\ \vdots & \ddots & \vdots \\ t_{D,1} & \cdots & t_{D,T} \end{bmatrix}. \tag{3.1}$$

Paso 3: Calcular los valores intermedios hipotéticos. El siguiente paso de ataque es calcular un valor intermedio hipotético para cada posible elección de k. Estos posibles valores son denotados por el vector $k = (k_1, ..., k_K)$, donde K denota el número total de posibles elecciones de k. En el contexto de un ataque los elementos del vector forman la clave secreta hipotética. Dado el vector d y la clave hipotética k, un atacante puede calcular fácilmente los valores intermedios hipotéticos $f(d, k)$ para todas las D operaciones de cifrados y para todas las K claves. Los valores se almacenan en una matriz $\mathbf{V}$:

$$\mathbf{V} = \begin{bmatrix} f(d_1, k_1) & \cdots & f(d_0, k_K) \\ \vdots & \ddots & \vdots \\ f(d_D, k_1) & \cdots & f(d_D, k_K) \end{bmatrix}. \tag{3.2}$$

Paso 4: Asignar valores intermedios a valores de consumo de potencia. El siguiente paso de ataque es mapear los valores intermedios hipotéticos V a la matriz $\mathbf{H}$ de valores de consumo de potencia hipotéticos. Para este propósito el atacante utiliza algún modelo de consumo de potencia. Al usar estas técnicas el consumo de potencia del dispositivo, cada uno de los valores intermedios hipotéticos $V_{i,j}$ es simulado en orden para obtener valores de consumo de potencia hipotéticos $H_{i,j}$. Por ejemplo si se utiliza el modelo de la distancia de Hamming ($HD(\cdot)$) la matriz $\mathbf{H}$ es:

$$\mathbf{H} = \begin{bmatrix} h_{1,1} & \cdots & h_{1,K} \\ \vdots & \ddots & \vdots \\ h_{D,1} & \cdots & h_{D,K}) \end{bmatrix} = \begin{bmatrix} HD(f(d_1, k_1)) & \cdots & HD(f(d_0, k_K)) \\ \vdots & \ddots & \vdots \\ HD(f(d_D, k_1)) & \cdots & HD(f(d_D, k_K)) \end{bmatrix}. \tag{3.3}$$

Paso 5: Comparar valores de consumo de potencia hipotéticos con trazas de potencia. Después de haber transformado (V) a $\mathbf{H}$, el paso final del ataque puede ser realizado. En este paso, cada columna h_i de la matriz H es comparado con cada columna t_j de la matriz T.Esto significa que el atacante compara los valores de consumo de potencia hipotético de cada una de las claves hipotéticas con las trazas registradas en cada posición. En esta comparación se utilizan los distinguidores, por ejemplo

[1] Se refiere al número de puntos capturados por un osciloscopio, esto depende de la frecuencia de muestreo del mismo y el periodo de tiempo de captura.

basándose en los coeficientes de correlación. A partir de esta relación, $f - bits$ de la clave secreta del dispositivo atacado puede ser descubierta con cierta probabilidad.

El procedimiento anterior es repetido varias veces con diferentes partes de los D mensajes conocidos, de esta manera se recupera la clave en su totalidad.

3.6. Plataforma experimetal

Para realizar un ACL utilizando el consumo de potencia se necesitan de los siguientes componentes:

1. Dispositivo objetivo (*target*): Es un dispositivo el cual va a ejecutar el algoritmo criptográfico que se quiere probar su seguridad.

2. Equipo de medición: Se encarga de recolectar las muestras, trazas para su posterior análisis. Se necesita de un osciloscopio que registre las señales durante la ejecución del algoritmo criptográfico.

3. Equipo de procesamiento: Con las señales medidas se realizan las correlaciones correspondientes con el consumo teórico contra las mediciones obtenidas. Con dicho análisis se puede obtener la clave utilizada en el algoritmo criptográfico.

Realizar estas pruebas, requieren de un osciloscopio de alta precisión, hardware adicional que realice la sincronización al ejecutar el algoritmo entre el dispositivo objetivo y el osciloscopio.

Por ello se utilizan herramientas dedicadas las cuales hacen el estudio de la seguridad de algoritmos criptográficos contra ataques de canal lateral asequible y simple de configurar.

Dichas herramientas de hardware y software son proporcionadas por NewAE[2].

NewAE es una empresa dedicada a la seguridad de hardware en dispositivos embebidos, con la misión de de hacer ver a los diseñadores e ingenieros sobre el poder del ACL y glitching como vectores de ataque importantes.

Las herramientas son de código abierto y ampliamente disponibles [36].

Se puede analizar distintas señales al mismo tiempo con un solo dispositivo, por ejemplo, consumo de potencia con emisiones electromagnéticas, sin necesidad de tener un equipo de captura para cada señal generada.

3.6.1. Ecosistema ChipWhisperer

Es un conjunto de herramientas útiles para la investigación en seguridad en hardware de dispositivos embebidos.

Están enfocadas a el análisis de consumo de potencia, voltaje y ataque de fallas del reloj.

[2] https://www.newae.com

Estas herramientas dan soporte a distintos dispositivos objetivo como microcontroladores ARM Cortex-M, dispositivos PowerPC, FPGAs pequeños.

Están compuestas de cuatro capas.

- Hardware: ChipWhisperer tiene tarjetas-osciloscopio que son utilizadas para montar un ACL.

  Las tarjetas funcionan como dispositivos bajo prueba.

- Firmware: Incluyen el firmware de código libre para las tarjetas-osciloscopio y los dispositivos objetivo.

  El firmware está escrito en Veriog para los FPGAs y en C para los microcontroladores.

- Software: Es una biblioteca de código abierto escrita en python que controla el hardware de captura de las trazas y la comunicación entre el dispositivo objetivo. Se encarga de obtener las mediciones del consumo de potencia. Tiene dos APIs principales una Captura y otra Analizer.

- Tutorials: Son libretas de python que funcionan para montar un laboratorio de ACL. Desde las libretas se puede realizar todo el flujo del ataque hasta realiza el análisis de las muestras capturadas y obtener la clave utilizada por el algoritmo criptográfico.

Las 4 capas siguen un flujo general para la captura hasta obtener la clave. Inicia con la configuración de la tarjeta-osciloscopio y el microcontrolador objetivo. Se escribe el texto plano o mensaje que va a procesar el algoritmo criptográfico dentro del microcontrolador. Después se sincronizan las dos partes y se activa el cifrado. A la vez se capturan las trazas. Se obtiene el cifrado del objetivo. Los datos obtenidos se organizan y se almacenan.

Estos pasos se realizan repetidamente hasta generar un conjunto de trazas con distintos textos planos cifrados suficientes para poder realizar el análisis, los resultados se interpretan aun si no se ha encontrado la clave y se pueden obtener resultados parciales de la clave.

3.6.2. ChipWhisperer Nano

Es la plataforma de menor costo para realizar ACL e inyección de fallos de voltaje. Tiene las siguientes características:

- Convertidor Analogíco-digital de 8 bits capaz de tomar muestras hasta 20 Megamuestras por segundo, con un reloj externo o un reloj interno.

- Tiene un microcontrolador objetivo STM32F030 para cargar el algoritmo criptográfico.

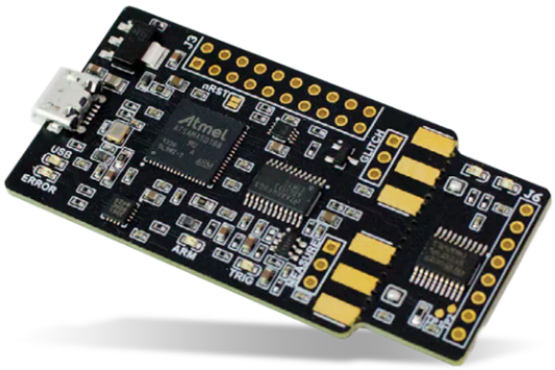


Figura 3.1: ChipWhisperer Nano, [37]

- Puede almacenar hasta 50 000 muestras.

La tarjeta ChipWhisperer Nano está diseñada principalmente para demostraciones de análisis de potencia y método de enseñanza. Físicamente la tarjeta está dividida en dos secciones una que es el microcontrolador y la que toma muestra.

3.6.3. ChipWhisperer Lite

De la misma manera que la plataforma ChipWhisperer Nano, la versión Lite también está dividida en dos módulos. Uno encargado de la captura de las muestras y el otro que implementa el algoritmo criptográfico a atacar. Esta versión tiene un mayor número de prestaciones que la versión Nano.

Esta versión tiene una mayor capacidad de almacenamiento de las trazas, opera a una frecuencia mayor y tienen una mayor sensibilidad a la señal de ruido.

Sus características son las siguientes:

- Convertidor Analógico-digital de 10 bits capaz de tomar muestras hasta 105 Mega-muestras por segundo, con un reloj externo o un reloj interno.
- Tiene un microcontrolador objetivo STM32F para cargar el algoritmo criptográfico.
- Un reloj ajustable de 5-200MHz.
- Puede almacenar hasta 24 573 muestras.

La herramienta de ChipWhisperer hacen que la configuración sea sencilla de ambas tarjetas. Basta con cambiar las opciones de compilación con una variable, por ello con los programas desarrollados para el ACL se puede comparar el funcionamiento en ambas plataformas.

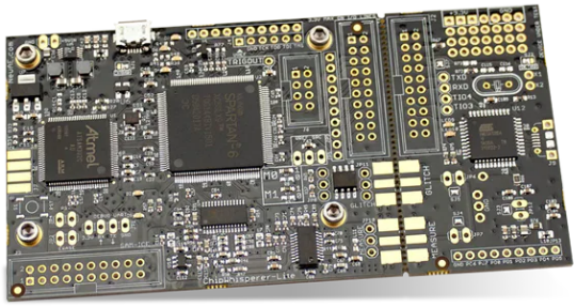


Figura 3.2: ChipWhisperer Lite, [3].

Capítulo 4

Implementación de los ataques

4.1. Implementación Secuencial

Nuestro análisis fue realizado tanto en GIFT-64 como GIFT-128, ya que independientemente de la versión, la clave está formada de 128 bits, en donde cada subclave es un *nibble*, lo que significa que cada subclave está compuesta de 4−bits y puede tener sus valores entre 0, 1, 2, ..., 15, es decir, 2^4 posibles combinaciones.

4.1.1. Modelo de consumo

Para poder llevar a cabo un ataque exitoso, es decir, obtener de manera correcta los 128 bits de la clave, necesitamos conseguir las 32 subclaves que la componen. El número de subclaves que se pueden extraer por ronda varía dependiendo de la versión analizada, ya que para GIFT-64 se utilizan 32 bits de la clave por ronda, es decir, ocho subclaves, por lo que es necesario atacar cuatro rondas de GIFT-64. Para GIFT-128 se utilizan 64 bits de la clave por ronda, obteniendo 16 subclaves, por lo que únicamente es necesario atacar dos rondas de GIFT-128.

Para cada una de las subclaves con la que se cifran N textos planos, dependiendo de la implementación atacada, calculamos un valor de correlación para cada posible valor [0,...,15], después de calcular los distintos valores de correlación, escogemos el valor de correlación máxima de cada subclave supuesta y ese será el valor escogido.

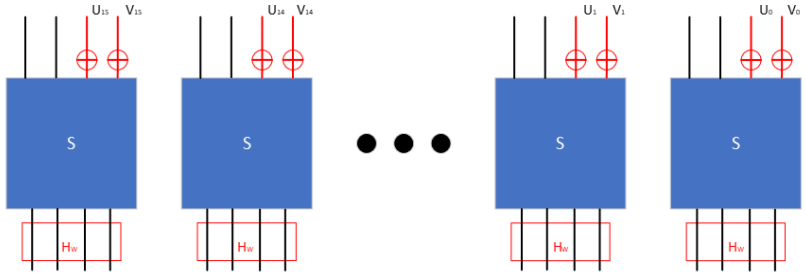


Figura 4.1: GIFT-64, Puntos de fuga de información

Este proceso es repetido en cada una de las rondas tanto de GIFT-64 como de GIFT-128.

En nuestro análisis elegimos el punto de interés (PdI), también conocido como punto de fuga en la salida de la *caja S* de la función de ronda 2, 3, 4 y 5 de GIFT-64 y en la salida de la *caja S* de la función de ronda 2 y 3 de GIFT-128.

La correlación es calculada para cada posible subclave supuesta usada en cada una de las rondas atacadas y la subclave con la correlación máxima se vuelve nuestra subclave supuesta. Ya que parte del contenido de la clave secreta es utilizada en cada ronda objetivo, para obtener la primer clave de ronda, calculamos el peso de Hamming de la salida de la *caja S* de la segunda porque es ahí donde existe una interacción entre el estado del cifrador y la clave de ronda. Para obtener las claves de ronda siguientes se repite el proceso con el fin de recuperar la clave secreta. El código mostrado abajo detalla el funcionamiento de la correlación.

Algoritmo 1 CPA [6]

```
1: for iteration = 1, 2, ... do
2:     for actor = 1, 2, ..., N do
3:         Clave supuesta XOR Texto plano
4:     end for
5:     Correlación Máxima
6: end for
```

4.1.2. Reconstrucción de la clave

La clave secreta también conocida como *clave de estado K* utilizada por el cifrador es de 128 bits como $K = k_7||k_6||...||k_0$ donde k es una palabra de 16 bits [24].

El calculo de las claves de ronda para ambas versiones de GIFT es el siguiente:

Para GIFT-64 se extraen dos palabras de 16 bits, k_0 y k_1 de la clave de estado y se convierten en la clave de ronda $RK = U||V$.

$$U \leftarrow k_1, V \leftarrow k_0$$

Para GIFT-128 se extraen cuatro palabras de 16 bits, k_0, k_1, k_4 *y* k_5 de la clave de estado y se convierten en la clave de ronda $RK = U||V$.

Después se actualiza la clave de estado como:

$$k_7||k_6||...||k_1||k_0 \leftarrow k_1 \gg 2||k_0 \gg 12||...||k_3||k_2$$

donde $\gg i$ es una rotación circular a la derecha.

Inversión *Key Schedule* de GIFT-64

Para obtener la clave secreta se necesitan las primeras cuatro claves de ronda y después utilizamos esas cuatro claves de ronda para invertir el proceso de generación de claves y recuperar los 128 bits que la componen. Necesitamos atacar cuatro rondas secuenciales de GIFT-64 debido a la estructura de su esquema criptográfico. En GIFT-64, únicamente 32 bits de cada clave de ronda se utilizan, es por lo que son necesarias cuatro rondas secuenciales para obtener los 128 bits de la clave secreta.

En la función de ronda *AddRoundKey* cada uno de los bits de U y V se combinan con el estado del cifrador mediante una operación XOR en los bits $b_{4i+1} y b_4$, por lo que es necesario generar U y V y obtener la clave de ronda [24].

$$b_{4i+1} \leftarrow b_{4i+1} \oplus u_i, b_{4i+1} \leftarrow b_{4i} \oplus v_i, \forall_i \in \{0, ..., 15\}$$

Inversión *Key Schedule* de GIFT-128

Para obtener la clave secreta se necesitan las primeras dos claves de ronda y después utilizamos esas dos claves de ronda para invertir el proceso de generación de claves y recuperar los 128 bits que la componen. Necesitamos atacar dos rondas secuenciales de GIFT-128 debido a la estructura de su esquema criptográfico. En GIFT-128, únicamente 64 bits de cada clave de ronda se utilizan, es por lo que son necesarias dos rondas secuenciales para obtener los 128 bits de la clave secreta.

En la función de ronda *AddRoundKey* cada uno de los bits de U y V se combinan con el estado del cifrador mediante una operación XOR en los bits b_{4i+2} *y* b_{4+1}, por lo que es necesario generar U y V y obtener la clave de ronda [24].

$$b_{4i+2} \leftarrow b_{4i+2} \oplus u_i, b_{4i+1} \leftarrow b_{4i+1} \oplus v_i, \forall_i \in \{0, ..., 31\}$$

4.1.3. Configuración del ataque

Para este análisis se utilizó el ecosistema ChipWhisperer que cuenta con un microcontrolador que tiene como objetivo ser la víctima, la tarjeta de control y el osciloscopio. El modelo utilizado fue el ChipWhisperer NANO que consiste en un microcontrolador STM32F030 de ocho bits. La implementación utilizado tanto de GIFT-64 como de GIFT-128 fue implementada en el lenguaje C con la ayuda del protocolo simple serial de ChipWhisperer. Se realizaron una serie de ataques para poder visualizar la tasa de efectividad del modelo propuesto, esto significa, el número de veces que se obtuvo con éxito la clave secreta. Para poder visualizar la efectividad del ataque, se midió la tasa de éxito del ataque de potencia correlacional (APC), está tasa de éxito muestra si un ataque recuperó de manera exitosa la clave usada por el algoritmo para cifrar los textos planos. La tasa de éxito se definió como:

$$\text{tasa de éxito} = \begin{cases} 1, & \text{si la clave es correcta} \\ 0, & \text{de lo contrario} \end{cases}$$

Para realizar el análisis de la efectividad del ataque, se realizaron múltiples ataques y se calculó la media de la tasa de éxito, que es el número de ataques intentados dividido por el número de ataques que recuperaron la clave de manera exitosa.

$$media = \frac{\#\ ataques\ exitosos}{\#\ total\ ataques}$$

Una serie de ataques es definida como un conjunto de iteraciones, en donde cada iteración crea de manera aleatoria un arreglo de pares de textos planos y la potencia medida y se guarda en un bloque de datos para posteriormente aplicar el ataque sobre estos datos. Si el ataque recuperó la clave secreta de manera exitosa, la el valor de la tasa de éxito es 1 de lo contrario es 0. Una vez que las iteraciones se terminan, la media de la tasa de éxito es calculada. En nuestro análisis se usaron un total de 120 iteraciones y 5000 pares de textos planos y la potencia medida. El procedimiento es descrito en el algoritmo 2.

Algoritmo 2 Serie de ataques APC

```
1: Resultado: Media de la tasa de éxito
2: totalIteraciones = 120
3: iteraciones = 0
4: tasaExito = 0
5: mediaTasaExito = 0
6: pares = []
7: while totalIteraciones < total do
8:     pares.agregar(texto/potencia)
9:     resultado = CPA(pares)
10:    iteraciones++
11:    if resultado es exitoso then
12:        tasaExito += 1
13:    end if
14: end while
15: mediaTasaExito = tasaExito/totalIteraciones
```

4.2. Implementación bitslicing

En ambas versiones de GIFT-64 y GIFT-128 se usa la representación de una matriz para realizar paralelamente las operaciones de la caja S. Sin embargo la parte costosa a nivel software es la permutación de bits. Para que la entrada concuerde con el formato de la matriz, este debe transformarse con las función de `packing` y el cifrado se regresa a la representación normal `unpacking`. Para ello se utiliza la siguiente función de intercambio de bits `SWAPMOVE` Algoritmo 3.

En ambas versiones de GIFT-64 y GIFT-128 el mensaje debe de transformarse para seguir el mismo formato, los siguientes pseudocódigos ilustran esta acción Algo-

Algoritmo 3 SWAPMOVE

Require: $a, b, mask, n$ ▷ Dos números a y b de 32 bits, un desplazamiento y una máscara

1: $\text{tmp} \leftarrow (b \oplus (a \gg n)) \wedge \text{mask}$
2: $b \leftarrow b \oplus \text{tmp}$
3: $a \leftarrow a \oplus (\text{tmp} \ll n)$

ritmo 12. El unpacking se utilizan las mismas operaciones solo que la transformación de enteros de 32 bits en formato Big-endian a un arreglo de bytes se realiza al final de los intercambios de bits. Debe de tomarse en cuenta que se trabaja con un microcontrolador ARM Cortex-M de 32 bits, generalmente están configurados para trabajar con la representación Little-endian (el byte menos significativo se almacena en la memoria más baja). La permutación fue implementada de la siguiente manera, utilizando la función de intercambio `SWAP`.

4.2.1. Modelo de consumo

Peso de Hamming

Se implementó el peso de Hamming como modelo de consumo para los ataques de correlación de potencia . Específicamente, se consideraron las transiciones de 1 y 2 bits en los Algoritmos 4 y 5 en las operaciones de cifrado. Esto significa que el modelo de consumo se basa en el número de bits que cambian a "1.[en] cada operación. Para la implementación, se calculó el peso de Hamming de cada byte procesado en el cifrado y se utilizó esta información para correlacionarla con las mediciones de consumo de potencia del dispositivo objetivo. Esta implementación permitió evaluar cómo las variaciones en el consumo de energía pueden estar asociadas con el comportamiento de GIFT-COFB en sus versiones de 64 y 128-bits, en particular con los valores intermedios de las claves.

Algoritmo 4 Cálculo del peso de Hamming de 1 bit

Require: entrada: arreglo de bytes para el cálculo del peso de Hamming

1: peso_de_hamming ← 0 ▷ Inicializa el peso de Hamming
2: **for** byte ∈ entrada **do**
3: **for** $i \leftarrow 0$ to 7 **do**
4: **if** pos_bi(byte, i) = 1 **then**
5: peso_de_hamming ← peso_de_hamming + 1
6: **end if**
7: **end for**
8: **end for**
9: **return** peso_de_hamming ▷ Retorna el peso de Hamming total de 1 bit

Algoritmo 5 Cálculo del peso de Hamming de 2 bits consecutivos

Require: entrada: arreglo de bytes para el cálculo del peso de Hamming
1: peso_de_hamming_2bits ← 0 ▷ Inicializa el peso de Hamming de 2 bits
2: **for** byte ∈ entrada **do**
3: **for** $i \leftarrow 0$ to 6 **do**
4: **if** pos_bit(byte, i) = 1 and pos_bit(byte, i+1) = 1 **then**
5: peso_de_hamming_2bits ← peso_de_hamming_2bits + 1
6: **end if**
7: **end for**
8: **end for**
9: **return** peso_de_hamming_2bits ▷ Retorna el peso de Hamming de 2 bits

4.2.2. SNR

En este análisis de potencia, se utilizó SNR como modelo de consumo para los ataques de análisis de potencia diferencial DPA y análisis de potencia de correlación CPA. Específicamente, el SNR Algoritmo 6 se aprovecha para identificar la diferencia entre la señal útil, que corresponde a las variaciones en el consumo de potencia asociadas con las operaciones de cifrado, y el ruido de fondo que puede distorsionar la medición.

4.2.3. Reconstrucción de la clave

En los ataques de CPA la reconstrucción de la clave se basa en correlacionar las mediciones de consumo de energía con los valores intermedios generados durante el proceso criptográfico. En CPA, se utilizó el peso de Hamming y el SNR como modelos de consumo para predecir el consumo de energía asociado con diferentes valores de la clave Algoritmo 7. A través de la correlación entre las trazas y las predicciones del modelo, se pudieron identificar los valores correctos de la clave. En DPA, se calculó la diferencia entre las trazas asociadas a diferentes hipótesis de la clave y se utilizó un análisis estadístico para detectar patrones significativos que revelaran la clave correcta. Ambos ataques requerían múltiples trazas y un análisis cuidadoso de la variabilidad del consumo de energía para finalmente reconstruir la clave completa.

Previamente se utiliza la función `gift_internal` (Algoritmo 8 para realizar una operación para que las claves propuestas tengan el mismo formato que el GIFT-COFB. La función realiza una operación XOR entre el valor en la posición indicada por `row` y un valor basado en la suposición. La suposición se modifica según su paridad y su valor absoluto.

Se necesitan las siguientes subrutinas para reconstruir las claves:

- `construir_clave (Algoritmo 9)`: Esta función toma una lista de bits y construye una clave al aplicar un desplazamiento (*shift*) y una operación XOR entre los bits, generando un valor entero final que representa la clave reconstruida.

Algoritmo 6 Cálculo de la relación señal-ruido (SNR)

Require: ronda: entradas del estado del cifrado, trazas: conjunto de trazas de potencia, clave: clave secreta, fila: índice del estado a analizar, inicio: inicio del intervalo de trazas, fin: fin del intervalo de trazas

Ensure: Relación señal-ruido (SNR)

Inicializar grupos de peso de Hamming vacíos: grupos_hw ← []
for $i = 0$ **to** 16 **do**
    grupos_hw[i] ← []
end for
for $i = 0$ **to** longitud(trazas) − 1 **do**
    $peso\_hamming$ ← calcular_HW(estado_interno(ronda[i], clave, fila))
    Agregar trazas[i][inicio:fin] a grupos_hw[peso_hamming]
end for
Filtrar los grupos no vacíos: grupos_no_vacíos ← filtrar_no_vacíos(grupos_hw)
Calcular los promedios de cada grupo: promedios_hw ← calcular_promedios(grupos_no_vacíos)
Calcular el promedio global: promedio_global ← promedio(promedios_hw)
Inicializar el arreglo de ruido: ruido ← []
for $i = 0$ **to** longitud(promedios_hw) − 1 **do**
    for traza **in** grupos_no_vacíos[i] **do**
        Agregar traza − promedios_hw[i] a ruido
    end for
end for
Calcular la varianza del ruido: varianza_ruido ← calcular_varianza(ruido)
Calcular la varianza del promedio: varianza_promedio ← calcular_varianza(promedios_hw)
return $\frac{\text{varianza\_promedio}}{\text{varianza\_ruido}}$

Algoritmo 7 Análisis de correlación de potencia (CPA)

Require: rondas: lista de entradas del estado del algoritmo, trazas: conjunto de mediciones de potencia, registro: índice del estado a evaluar
Ensure: claves: lista de posibles valores de la clave

claves ← []
promedio_trazas ← calcular_promedio(trazas)
desv_trazas ← calcular_desviación(trazas, promedio_trazas)
$N \leftarrow 16$ ▷ Número de posibles bits de la clave
for bit ← 0 **to** $N - 1$ **do**
    maxcpa ← [0, 0] ▷ Almacenar las correlaciones máximas
    for suposición ← 0 **to** 1 **do**
        hws ← calcular_pesos_Hamming(rondas, bit, suposición, registro)
        promedio_hws ← calcular_promedio(*hws*)
        desv_hws ← calcular_desviación(*hws*, promedio_hws)
        correlacin ← calcular_covarianza(trazas, promedio_trazas, *hws*, promedio_hws)
        cpa_salida ← $\frac{\text{correlación}}{\text{desv\_trazas} \times \text{desv\_hws}}$
        maxcpa[suposición] ← calcular_máximo(valor_absoluto(cpa_salida))
    end for
    suposición_máxima ← indice_máximo(maxcpa)
    correlación_máxima ← máximo(*maxcpa*)
    Agregar suposición_máxima a claves
end forreturn claves

Algoritmo 8 gift_internal

Require: state: arreglo que representa el estado del sistema.
Require: guess: suposición sobre el valor a ser calculado.
Require: row: índice de la fila sobre la que se aplicará la operación.

return state[row] $\oplus$ ((guess mód 2) $\ll$ int(guess/2))

- `obtener_clave (Algoritmo 10)`: Esta función divide una lista de bits en dos grupos y luego utiliza la función `construir_clave` para calcular dos claves, una para cada grupo de bits.

Algoritmo 9 construir_clave

Require: bits: lista de bits de entrada
Ensure: Una clave generada a partir de los bits
1: $clave \leftarrow 0$
2: **for** (i, bit) **in** enumerar(bits) **do**
3: $\quad clave \leftarrow$ clave $\oplus$ (bit $\ll i$)
4: **end for**
5: **return** clave

Algoritmo 10 obtener_clave

Require: bits: lista de bits de entrada
Ensure: Dos claves generadas a partir de los grupos de bits
1: $grupo\_1 \leftarrow []$
2: $grupo\_2 \leftarrow []$
3: **for** $(bit1, bit2)$ **in** bits **do**
4: $\quad grupo\_1$.append(bit1)
5: $\quad grupo\_2$.append(bit2)
6: **end for**
7: **return** construir_clave(grupo_1), construir_clave(grupo_2)

Ataque CPA utilizando subclaves

El Algoritmo 11 implementa una variante del ataque CPA (Análisis de Correlación de Potencia) para identificar la clave más probable a partir de un conjunto de subclaves. La función `max_cpa` calcula la correlación entre las trazas de potencia y las subclaves posibles utilizando un modelo de consumo basado en el peso de Hamming. La subclave que produce la mayor correlación con las trazas observadas es seleccionada como la clave más probable.

En este algoritmo se implementan dos funciones (Algoritmos 12 y 13, `packing` y `unpacking`, que se utilizan para transformar los textos planos y cifrados en el formato `GIFT-COFB`. La función `packing` toma un bloque de texto plano y lo organiza en un formato adecuado para ser procesado por el cifrado. La función `unpacking`, por otro lado, toma un bloque cifrado y lo convierte de nuevo al formato de texto plano. Ambas funciones utilizan la operación `SWAPMOVE` para realizar intercambios de bits dentro del estado.

Algoritmo 11 max_cpa

Require: ronda: conjunto de valores del estado del cifrado
Require: trazas: trazas de potencia capturadas
Require: subclaves: lista de subclaves posibles
Require: fila: índice de la fila del estado a evaluar
Ensure: La subclave más probable

```
promedio_trazas ← calcular_promedio(trazas)
desv_trazas ← calcular_desviación(trazas, promedio_trazas)
maxcpa ← [0] × longitud(subclaves)
for suposición in rango(0, longitud(subclaves)) do
    hws ← calcular_pesos_Hamming(ronda, subclaves[suposición], fila)
    promedio_hws ← calcular_promedio(hws)
    desv_hws ← calcular_desviación(hws, promedio_hws)
    correlacin ← calcular_covarianza(trazas, promedio_trazas, hws, promedio_hws)
    cpa_resultado ← correlación / (desv_trazas × desv_hws)
    maxcpa[suposición] ← calcular_máximo(valor_absoluto(cpa_resultado))
end for
return subclaves[índice_máximo(maxcpa)]
```

Algoritmo 12 packing

Require: block: arreglo de entrada de 8 bytes

```
state ← [0] * 4
state[0] ← U32BIG(block[4 :])
state[1] ← U32BIG(block[: 4])
SWAPMOVE(0, 0, 0x0a0a0a0a, 3)
SWAPMOVE(0, 0, 0x00cc00cc, 6)
SWAPMOVE(0, 0, 0x0000f0f0, 12)
SWAPMOVE(0, 0, 0x0000ff00, 8)
SWAPMOVE(1, 1, 0x0a0a0a0a, 3)
SWAPMOVE(1, 1, 0x00cc00cc, 6)
SWAPMOVE(1, 1, 0x0000f0f0, 12)
SWAPMOVE(1, 1, 0x0000ff00, 8)
SWAPMOVE(0, 1, 0x00ff00ff, 8)
SWAPMOVE(0, 2, 0x0000ffff, 16)
SWAPMOVE(1, 3, 0x0000ffff, 16)
return state
```

Algoritmo 13 unpacking

Require: state: arreglo de estado de 4 palabras

```
SWAPMOVE(1, 3, 0x0000ffff, 16)
SWAPMOVE(0, 2, 0x0000ffff, 16)
SWAPMOVE(0, 1, 0x00ff00ff, 8)
SWAPMOVE(1, 1, 0x0000ff00, 8)
SWAPMOVE(1, 1, 0x0000f0f0, 12)
SWAPMOVE(1, 1, 0x00cc00cc, 6)
SWAPMOVE(1, 1, 0x0a0a0a0a, 3)
SWAPMOVE(0, 0, 0x0000ff00, 8)
SWAPMOVE(0, 0, 0x0000f0f0, 12)
SWAPMOVE(0, 0, 0x00cc00cc, 6)
SWAPMOVE(0, 0, 0x0a0a0a0a, 3)
```

block[4 :] ← U8BIG(state[0])
block[: 4] ← U8BIG(state[1])
return block

4.2.4. Flujo completo del ataque

Este conjunto de algoritmos representa el flujo completo de un ataque de análisis de potencia diferencial (DPA) utilizando correlación (CPA). El proceso comienza con la captura de trazas de consumo de potencia (Algoritmo 14). En este paso, se configura un dispositivo criptográfico para cifrar textos planos generados de forma aleatoria, utilizando una clave secreta fija. Durante el cifrado, se registran las trazas de potencia que reflejan el consumo energético del dispositivo, junto con los textos de entrada utilizados. Este conjunto de trazas y textos es fundamental, ya que proporciona la base de datos necesaria para realizar el ataque.

Una vez capturadas las trazas, el siguiente paso es recuperar la clave secreta mediante un ataque CPA (Algoritmo 15). Primero, se identifican las partes más relevantes de las trazas mediante un análisis de la relación señal-ruido (SNR). Esto ayuda a localizar los puntos donde las trazas contienen información útil sobre la clave. Luego, se extraen esas subtrazas y se aplica el análisis de correlación de potencia para deducir las partes probables de la clave secreta. Estas partes se refinan a través de cálculos adicionales, como la búsqueda de correlaciones máximas, y finalmente se reconstruye la clave completa en el formato que utiliza el algoritmo de cifrado.

En resumen, estos algoritmos describen el flujo completo de un ataque, desde la configuración inicial para capturar las trazas hasta la recuperación precisa de la clave secreta. Este enfoque puede adaptarse a diferentes rondas del cifrado o incluso a otros algoritmos criptográficos, haciendo del proceso una herramienta flexible para el análisis de seguridad.

Algoritmo 14 Captura de trazas de potencia

Require: N: Número total de trazas a capturar, clave: clave del algoritmo de cifrado
Ensure: arreglo_trazas: conjunto de trazas capturadas, arreglo_textos: textos de entrada correspondientes

```
arreglo_trazas ← []
arreglo_textos ← []
iniciar_cifrado(clave)                          ▷ Inicializar el sistema con la clave
texto_cifrado_tmp ← None
for i ← 1 to N do
    preparar_captura()                          ▷ Preparar sistema para capturar traza
    texto ← generar_texto_aleatorio()           ▷ Generar texto plano aleatorio
    enviar_texto(texto)                         ▷ Enviar texto plano al dispositivo para cifrar
    esperar_cifrado()                           ▷ Esperar a que el dispositivo complete el cifrado
    if cifrado_agotado() then
        continue
    end if
    respuesta ← leer_salida_cifrada()           ▷ Leer texto cifrado
    if i = 1 then
        texto_cifrado_tmp ← respuesta
    end if
    traza ← obtener_traza_actual()              ▷ Obtener traza de consumo
    arreglo_trazas.append(traza)
    arreglo_textos.append(texto)
end for
return arreglo_trazas, arreglo_textos
```

Algoritmo 15 Obtención de claves probables mediante CPA

Require: trazas: conjunto de trazas capturadas, textos: textos planos correspondientes, rondas: datos intermedios calculados por el cifrado
Ensure: clave_recuperada: clave secreta reconstruida
1: snr_graficar(rondas, trazas, región_interes) ▷ Identificar región relevante
2: trazas_primera_ronda ← extraer_trazas(trazas, región_interes) ▷ Extraer subtrazas relevantes
3: parte_v ← cpa(rondas, trazas_primera_ronda, registro_v) ▷ Calcular primera parte de la clave
4: parte_u ← cpa(rondas, trazas_segunda_ronda, registro_u) ▷ Calcular segunda parte de la clave
5: v_0, v_1 ← formato_clave(parte_v)
6: u_0, u_1 ← formato_clave(parte_u)
7: v1 ← max_cpa(rondas, trazas_primera_ronda, [v_0, v_1], registro_v) ▷ Encontrar valor más probable de v
8: u1 ← max_cpa(rondas, trazas_segunda_ronda, [u_0, u_1], registro_u) ▷ Encontrar valor más probable de u
9: k12, k13, k14, k15 ← reconstruir_clave(v1, u1) ▷ Convertir a formato final
10: **return** k12, k13, k14, k15

4.3. Implementación fixslicing

4.3.1. Modelo de consumo

La implementación *fixslicing* de GIFT-COFB introduce una reorganización específica en la representación del estado, optimizada para realizar operaciones eficientes. Sin embargo, los ataques descritos anteriormente, como el cálculo de la Relación Señal-Ruido (SNR) y el Ataque de Correlación de Potencia (CPA), siguen siendo aplicables con ciertas adaptaciones necesarias para manejar esta representación.

En *fixslicing*, el estado del cifrado se reorganiza de manera que cada bit ocupa una posición fija, facilitando operaciones como permutaciones y mezclas en un nivel estructurado. Esto implica:

- Los bits del estado no están distribuidos secuencialmente, sino que están asignados de manera fija y específica a registros o palabras de la arquitectura subyacente.
- Las operaciones internas del cifrado, como las permutaciones de bits, las rotaciones y las mezclas, se realizan de manera diferente al enfoque clásico o al *bit-slicing*.

Dado esto, las métricas de análisis, como el peso de Hamming, deben ajustarse para reflejar esta representación. Las posiciones fijas de los bits determinan cómo se calcula el modelo de consumo y cómo se agrupan las trazas para analizar la relación entre los datos internos del cifrado y las mediciones de potencia.

El modelo basado en el peso de Hamming permanece sin cambios. Se asume que la potencia consumida por el hardware está directamente relacionada con el número de bits activados (en estado '1'). Sin embargo, debido a la representación *fixslicing*, se debe reinterpretar el estado para calcular correctamente este peso, teniendo en cuenta las posiciones fijas de los bits.

El cálculo de la SNR se realiza de manera similar al caso tradicional, pero los intervalos de trazas y la agrupación de pesos de Hamming deben adaptarse para considerar cómo están organizados los bits en *fixslicing*. En particular:

- La función de agrupamiento del peso de Hamming debe interpretar correctamente la posición fija de cada bit en el estado.
- Las trazas se analizan en intervalos definidos por las operaciones internas del cifrado, ajustándose a cómo las posiciones fijas de los bits afectan el modelo de consumo.

El Ataque CPA sigue el mismo principio de correlación entre las trazas de potencia y los valores intermedios del cifrado, pero es fundamental ajustar las funciones internas para interpretar correctamente la representación *fixslicing*:

- La función interna del cifrado (`gift_internal`) debe considerar la disposición fija de los bits al calcular los pesos de Hamming.
- La correlación se evalúa de forma estándar, pero el significado de cada bit dentro del estado debe alinearse con la estructura de *fixslicing*.

4.3.2. Reconstrucción de las claves

El proceso de reconstrucción de claves, basado en las subrutinas descritas previamente, también debe ajustarse a la representación *fixslicing*. Las operaciones de desplazamiento y XOR se aplican de acuerdo con el esquema fijo de los bits, asegurando que el resultado final respete la estructura interna del estado. A pesar de estas adaptaciones, el flujo general para reconstruir las subclaves y la clave completa permanece intacto.

En la implementación *fixslicing* de GIFT-COFB:

- El cálculo de SNR y CPA es funcional, pero requiere ajustes para trabajar con la representación fija de los bits.
- Las métricas, como el peso de Hamming, se deben calcular considerando las posiciones específicas de los bits en *fixslicing*.
- Las correlaciones y las trazas capturadas deben interpretarse correctamente para reflejar la estructura fija del estado.

Las siguientes funciones están diseñadas para dar formato a las claves utilizadas en la representación *fixslicing* del cifrado GIFT-COFB. Esta representación requiere que las claves se estructuren de manera específica, reorganizando los bits para optimizar las operaciones internas, como transposiciones, rotaciones y manipulaciones bit a bit. A continuación, se describen y detallan las funciones utilizadas para lograr este objetivo.

Las funciones `REARRANGE_KEYWORD_0_1` y `REARRANGE_KEYWORD_2_3` procesan pares de palabras clave (`x` e `y`), aplicando desplazamientos y operaciones lógicas como `AND`, `OR` y `SHIFT`. Estas funciones generan combinaciones específicas de los bits para alinear las palabras clave al formato requerido por el *fixslicing*. La diferencia entre ambas radica en el esquema de combinación utilizado para procesar los bits.

La función principal, `RearrangeKey`, toma como entrada una clave organizada en palabras y la transforma en un conjunto de palabras clave reorganizadas (`rkey`). Esto se logra mediante una combinación de operaciones:

- Aplicación de `REARRANGE_KEYWORD_0_1` y `REARRANGE_KEYWORD_2_3` a las palabras clave individuales.
- Transposición de palabras mediante la subrutina `TRANSPOSE_U32`.
- Operaciones lógicas como `XOR`, `OR` y desplazamientos para completar la reorganización.
- Uso de la subrutina `SWAPMOVE`, que realiza intercambios controlados entre palabras clave utilizando máscaras y desplazamientos.

Estas operaciones aseguran que la clave reorganizada cumpla con los requisitos del formato *fixslicing* y esté lista para su uso en el cifrado.

En resumen, aunque los principios fundamentales de los ataques no cambian, su implementación práctica necesita tomar en cuenta los detalles particulares de *fixslicing*, especialmente en lo que respecta al ordenamiento y manipulación de los bits en el estado del cifrado, así como la reconstrucción de las claves.

Algoritmo 16 Reorganización de palabras clave para *fixslicing*

function REARRANGE_KEYWORD_0_1(x, y)
    return $(((y) \wedge 0xf0) \ll 20) \vee (((x) \wedge 0x0f) \ll 16) \vee (((x) \wedge 0xf0) \ll 4) \vee ((y) \wedge 0x0f)$
end function
function REARRANGE_KEYWORD_2_3(x, y)
    return $(((x) \wedge 0xf0) \ll 20) \vee (((x) \wedge 0x0f) \ll 16) \vee (((y) \wedge 0xf0) \ll 4) \vee ((y) \wedge 0x0f)$
end function
function REARRANGEKEY(key)
    $rkey \leftarrow [0] \times 10$
    $rkey[0] \leftarrow$ REARRANGE_KEYWORD_0_1$(key[14], key[15])$
    $rkey[1] \leftarrow$ REARRANGE_KEYWORD_0_1$(key[12], key[13])$
    $rkey[0] \leftarrow$ TRANSPOSE_U32$(rkey[0])$
    $rkey[1] \leftarrow$ TRANSPOSE_U32$(rkey[1])$
    $rkey[0] \leftarrow rkey[0] \vee (rkey[0] \ll 4)$
    $rkey[1] \leftarrow rkey[1] \vee (rkey[1] \ll 4)$
    $rkey[0] \leftarrow rkey[0] \oplus 0xffffffff$
    $rkey[2] \leftarrow$ REARRANGE_KEYWORD_0_1$(key[10], key[11])$
    $rkey[3] \leftarrow$ REARRANGE_KEYWORD_0_1$(key[8], key[9])$
    $rkey[2] \leftarrow rkey[2] \vee (rkey[2] \ll 4)$
    $rkey[3] \leftarrow rkey[3] \vee (rkey[3] \ll 4)$
    $rkey[2] \leftarrow rkey[2] \oplus 0xffffffff$
    SWAPMOVE(2, 2, 0x22222222, 2)
    SWAPMOVE(3, 3, 0x22222222, 2)
    $rkey[4] \leftarrow$ REARRANGE_KEYWORD_2_3$(key[6], key[7])$
    $rkey[5] \leftarrow$ REARRANGE_KEYWORD_2_3$(key[4], key[5])$
    $rkey[4] \leftarrow$ TRANSPOSE_U32$(rkey[4])$
    $rkey[5] \leftarrow$ TRANSPOSE_U32$(rkey[5])$
    SWAPMOVE(4, 4, 0x00000f00, 16)
    SWAPMOVE(5, 5, 0x00000f00, 16)
    $rkey[4] \leftarrow rkey[4] \vee (rkey[4] \ll 4)$
    $rkey[5] \leftarrow rkey[5] \vee (rkey[5] \ll 4)$
    $rkey[4] \leftarrow rkey[4] \oplus 0xffffffff$
    $rkey[6] \leftarrow$ REARRANGE_KEYWORD_2_3$(key[2], key[3])$
    $rkey[7] \leftarrow$ REARRANGE_KEYWORD_2_3$(key[0], key[1])$
    $rkey[6] \leftarrow rkey[6] \vee (rkey[6] \ll 4)$
    $rkey[7] \leftarrow rkey[7] \vee (rkey[7] \ll 4)$
    $rkey[6] \leftarrow rkey[6] \oplus 0xffffffff$
    $rkey[8] \leftarrow$ NIBBLE_ROR_1$(rkey[0])$
    $rkey[9] \leftarrow$ (NIBBLE_ROR_3$(rkey[1]) \wedge 0x0000ffff) \vee (rkey[1] \wedge 0xffff0000)$
    $rkey[9] \leftarrow$ ROR$(rkey[9], 16)$
    return $rkey$
end function

4.4. Estado del arte

Tabla 4.1: Ataques CPA a GIFT-COFB implementado en software

Autor	Version de GIFT-COFB	Plataforma	Herramienta	Num. de trazas	Resultado
Turan N. [38]	Bitslicing Protección mascara booleana	STM32F303	ChipWhisperer CW309	100k	falla prueba-t
Turan N. [38]	Sin protección	STM32F303RCT6	Osciloscopio Pico 3203D	20k	pasa prueba-t
Benjamin A. [39]	GIFT-128	STM32F	ChipWhisperer-Lite	345	100 % exactitud
Unger W.[40]	Sin protección	ATXMEGA128D4	ChipWhisperer CW308	2000	100 % exactitud

CPA es un método ampliamente utilizado para evaluar la seguridad de distintos criptosistemas, en este caso para GIFT-COFB frente a ataques de canal lateral. En un estudio realizado por [38], se evaluaron implementaciones de GIFT-COFB en un microcontrolador STM32F303 bajo dos configuraciones: una protegida mediante mascarado booleano y otra sin protección. Los resultados mostraron que, en la versión protegida, CPA no logró recuperar las claves privadas, incluso después de analizar 100,000 trazas, lo que evidencia una resistencia adecuada. Sin embargo, en la versión sin protección, CPA tampoco fue efectivo para comprometer la clave, pasando la prueba t con tan solo 20,000 trazas. Este comportamiento podría explicarse por las particularidades del formato *bitslicing* utilizado en GIFT-COFB, que complica la correlación directa entre las trazas de potencia y los estados intermedios de las claves.

Por otro lado, otros investigadores han demostrado que CPA puede ser altamente efectivo en ciertas condiciones. Por ejemplo, [40] consiguieron recuperar la clave con un 100 % de precisión analizando únicamente 2,000 trazas en un microcontrolador ATXMEGA128D4 utilizando un ChipWhisperer CW308. Tambien se han propuesto utilizar el CPA junto aprendizaje profundo por [39]. Encontraron que el número de trazas necesarias para vulnerar el cifrado se reducía considerablemente, a 345 para lograr una precisión de 100 %. Estos resultados subrayan la importancia de considerar tanto la representación fixslicing u otras métodos de protección contra CPA en GIFT y junto a su modo de operación.

En este trabajo se realiza un análisis CPA aplicado a la implementación fixslicing de GIFT-COFB. Este enfoque, que organiza los bits de manera específica para optimizar el rendimiento y reforzar la seguridad, representa un avance respecto a las investigaciones previas centradas en implementaciones *bitslicing*. Nuestro estudio es el primero en explorar las vulnerabilidades de GIFT-COFB bajo este esquema, lo que implica abordar retos únicos debido a la estructura y el ordenamiento particular de los bits en fixslicing. Esto requiere ajustar tanto el modelo de consumo como las técnicas de ataque para adaptarse a esta nueva representación.

Para garantizar que los resultados sean comparables con los de investigaciones anteriores, utilizamos las mismas herramientas estándar del estado del arte, como el ChipWhisperer, que permite la captura y evaluación precisa de trazas de potencia. Este enfoque asegura la consistencia de los datos y facilita una evaluación objetiva. Nuestro trabajo no solo amplía el conocimiento sobre las vulnerabilidades de GIFT-COFB, sino que también establece una base sólida para futuros análisis de cifrados livianos en esquemas más avanzados como el fixslicing.

Capítulo 5

Resultados

5.1. Implementación Secuencial

Para el análisis de la implementación secuencial del algoritmo GIFT se consideraró la versión de 64bits. El modelo de potencia que se eligió para atacar al algoritmo fue la correlación de Pearson junto con el peso de Hamming, esto debido a que la implementación usa la caja S con un bloque en memoria y procesa el estado de GIFT byte a byte. El punto clave aquí, es que cada caja S es calculada de manera separa por lo que el modelo de potencia es aplicada en el cálculo de cada una de ellas. Por lo tanto, es de esperarse que cada cuatro bits o *nibble* fugué información al mismo tiempo y sea posible suponer un consumo de potencia mediante el peso de Hamming de la salida de la caja S.

5.1.1. GIFT-64

El modelo propuesto se aplico a la salida de la caja S de la ronda 2, 3, 4 y 5 que es donde interactúan las claves de ronda respectivas con el estado del cifrador y se simularon los valores intermedios con las claves supuestas y se correlacionaron con la potencia medida. La Figura 5.1 muestra como se ven gráficamente una traza capturada de las rondas capturadas. Como se puede observar existe una región donde hay un patrón que se repite 28 veces, tal como el número de rondas que se aplican a la versión GIFT-64. El punto de interés se encuentra entre las trazas de 0 a 5000 ya que es donde se utiliza la clave, en especifico en las primeras rondas.

SNR

Para poder obtener el punto de fuga ó punto de interés para poder obtener las claves de ronda y de esta manera poder revertir el proceso que calcula las claves de ronda, se utilizó un algoritmo que muestra la tasa de señal a ruido y marca el punto en donde existe la mayor relación entre los valores intermedios y las trazas de potencia medidas. Estos posibles puntos de interés son visibles una vez que las trazas de potencia estén alineadas.

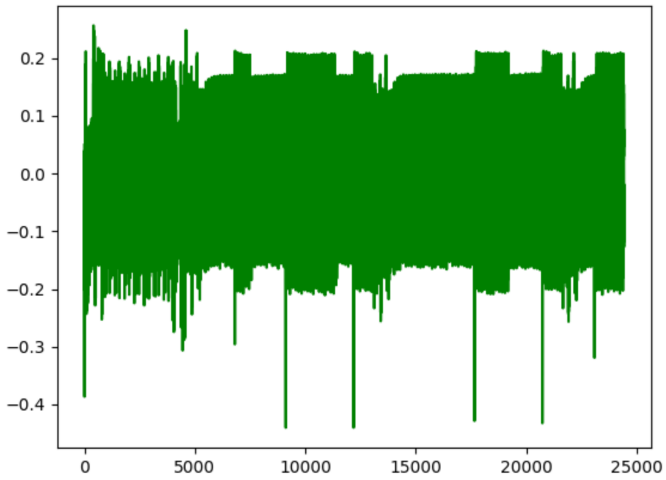
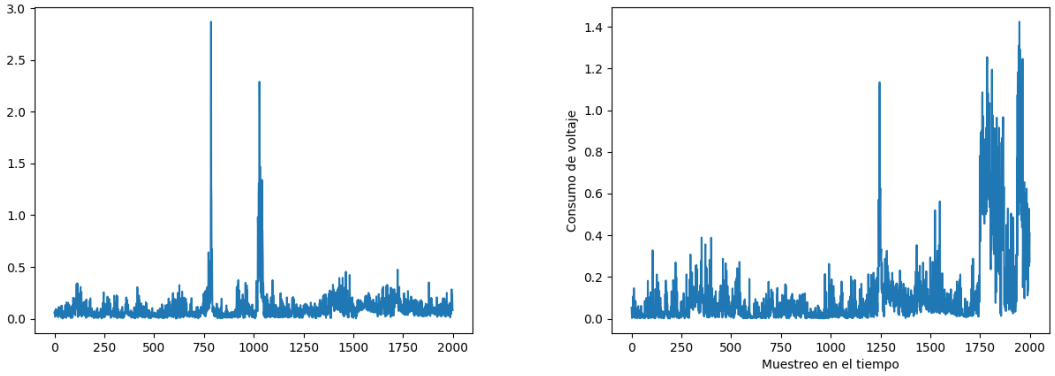


Figura 5.1: Traza capturada de GIFT-64 secuencial.

La Figura 5.2a muestra una gráfica con picos que muestran donde hay una mayor relación de la clave supuesta con los valores intermedios, donde el pico más alto señala el punto de interés que se utilizó para realizar la correlación de los valores supuestos con las trazas de potencia reales.



(a) Tasa de señal a ruido, ronda 1 y 2.

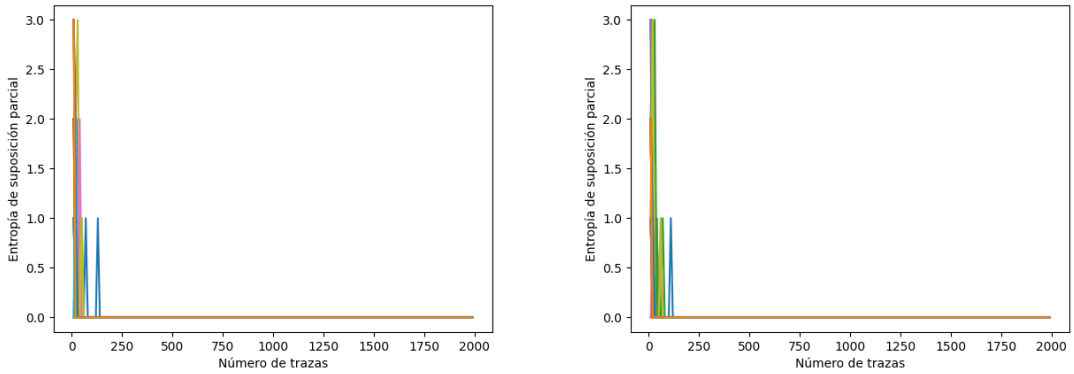
(b) Tasa de señal a ruido, ronda 4 y 5.

Figura 5.2: Gráficas de señal a ruido de rondas de la versión secuencial GIFT-64.

Correlación

El resultado del ataque en cada ronda es un conjunto de valores ordenados, en donde cada valor corresponde a la correlación máxima de las claves supuestas. El tamaño del conjunto de valores máximos varia dependiendo de la versión de GIFT analizada, siendo de 32 valores para GIFT-64 y 64 valores para GIFT-128. Cada uno de estos valores representa la subclave que se obtuvo como correcta.

La Figura 5.3a muestra que el número de trazas necesarias para obtener la primer clave de ronda es de 180. La Figura 5.3b muestra que le número de trazas necesarias para obtener la segunda clave de ronda es de 200.



(a) Entropía de suposición parcial, ronda 2. (b) Entropía de suposición parcial, ronda 2.

Figura 5.3: Gráficas de suposición de la cantidad de muestras necesarias para obtener la clave de la versión secuencial GIFT-64.

Lo mismo se replica para los rondas 4 y 5 y se obtienen las claves de ronda, tal como se muestra en la Figura5.2b.

5.1.2. GIFT-128

El análisis de GIFT-128 en la implementación secuencial se realizó siguiendo el mismo procedimiento que para GIFT-64. Se trabajó con las rondas 2, 3, 4 y 5, simulando los valores intermedios y correlacionándolos con las trazas de potencia obtenidas. Aunque no se logró generar gráficas específicas para esta versión, el análisis permitió confirmar que las fugas de potencia seguían un patrón similar al observado en GIFT-64, destacando los puntos clave donde las claves de ronda interactúan con el estado del cifrador. Los resultados de muestran en la Tabla 5.1.

5.2. Implementación bitslicing

Para el análisis de la implementación *bitslicing* del cifrador GIFT, se trabajó con las versiones de 64 bits y 128 bits. Esta implementación aprovecha un enfoque altamente paralelo que permite procesar múltiples bloques de datos al mismo tiempo. A diferencia de la versión secuencial, donde las operaciones se realizan de manera individual para cada *nibble*, en el *bitslicing* las operaciones se distribuyen a través de registros completos. Esto genera fugas de información de forma sincronizada, lo que introduce retos específicos al modelar el consumo de potencia. Para este análisis, se utilizó el modelo basado en la correlación de Pearson combinado con el peso de

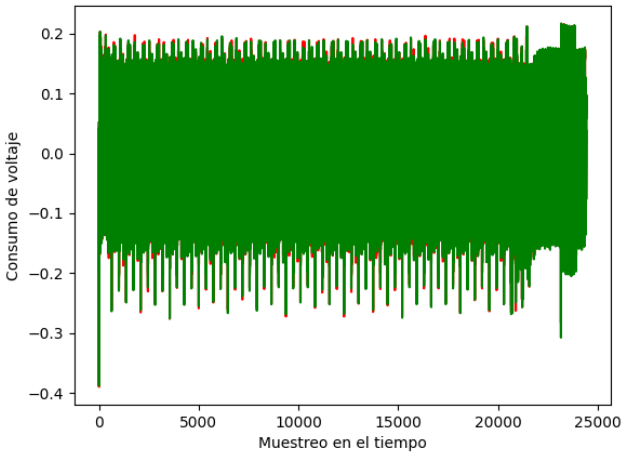


Figura 5.4: Traza capturada de GIFT-64 bitslicing.

Hamming, ya que sigue siendo efectivo para capturar las fugas relacionadas con las claves de ronda.

En esta implementación, las cajas S se procesan en paralelo, lo que significa que las transiciones de estado ocurren simultáneamente para múltiples bloques. Esto resulta en patrones distribuidos en las trazas de potencia, los cuales fueron analizados para identificar los puntos donde las claves interactúan con los estados intermedios.

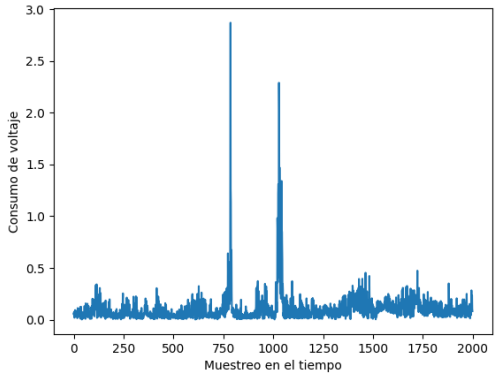
5.2.1. GIFT-64

En el caso de GIFT-64, se enfocó el análisis en las rondas 1,2,3 y 4, ya que en estas rondas las claves de ronda interactúan directamente con el estado del cifrador. Se utilizaron las trazas de potencia capturadas y se compararon con los valores intermedios calculados usando claves supuestas. En la Figura 5.4 se presenta una traza capturada, donde se puede observar un patrón repetitivo que corresponde a las 28 rondas de GIFT-64. El análisis de las fugas se concentró entre el tiempo 0 y 20000, dado que es en este rango donde ocurren las transiciones más significativas relacionadas con el uso de la clave.

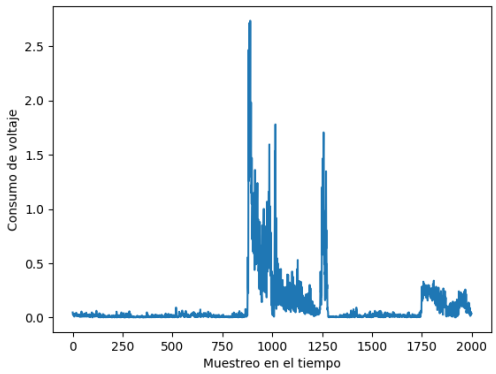
snr

Para identificar el punto de fuga o punto de interés que permitiera recuperar las claves de ronda y revertir el cálculo de estas, se utilizó un algoritmo basado en la SNR. Este algoritmo evalúa la relación entre los valores intermedios simulados y las trazas de potencia medidas, marcando los puntos con mayor correlación. Los puntos de interés se hacen visibles una vez que las trazas de potencia están correctamente alineadas, en este caso son se puede observar los picos de las rondas 1 a 4 en las figuras 5.5a5.5b5.5c5.5d. La Figura 5.5 muestra una gráfica donde los picos reflejan las zonas

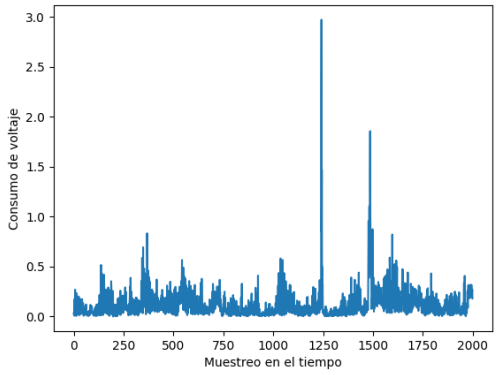
con mayor relación entre la clave supuesta y los valores intermedios en las distintas rondas. El pico más alto corresponde al punto de interés clave, que fue utilizado para realizar la correlación entre los valores supuestos y las trazas de potencia reales.



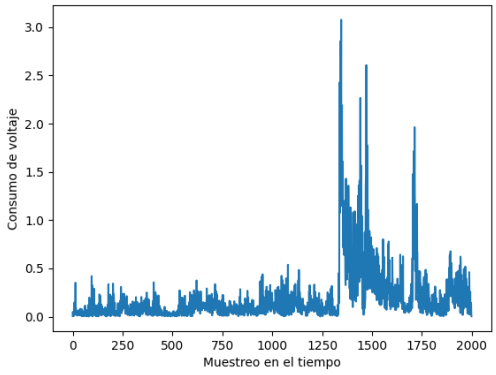
(a) Taza señal a ruido ronda 1



(b) Taza señal a ruido ronda 2



(c) Taza señal a ruido ronda 3



(d) Taza señal a ruido ronda 4

Figura 5.5: Gráficas de señal a ruido de rondas de la versión secuencial GIFT-64 *bitslicing*

5.2.2. GIFT-128

Para GIFT-128, el enfoque fue similar, pero con algunas diferencias debido a la estructura del cifrador, que opera sobre 40 rondas en lugar de 28. Las fugas de potencia también se analizaron en las rondas iniciales (1,2,3 y 4), donde las claves de ronda interactúan con el estado interno. En este caso, la Figura 5.6 muestra una traza capturada, en la que es evidente un patrón consistente con la estructura repetitiva de las rondas. Como en el caso de GIFT-64, el análisis se centró en las primeras 0 a

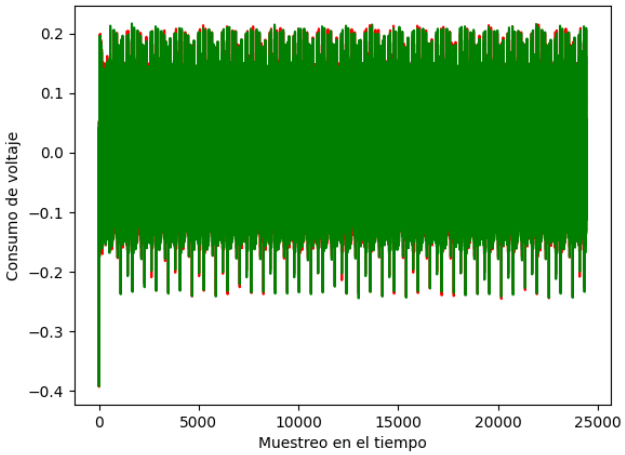


Figura 5.6: Traza capturada de GIFT-128 *bitslicing*.

25000 muestras, ya que es en este intervalo donde las transiciones relacionadas con las claves de ronda son más evidentes.

El procesamiento paralelo de GIFT-128 en *bitslicing* genera trazas más complejas debido al mayor número de operaciones en cada ronda. Sin embargo, los patrones de fuga son identificables al correlacionar las trazas con los modelos de potencia, permitiendo recuperar de manera efectiva las claves de ronda correspondientes.

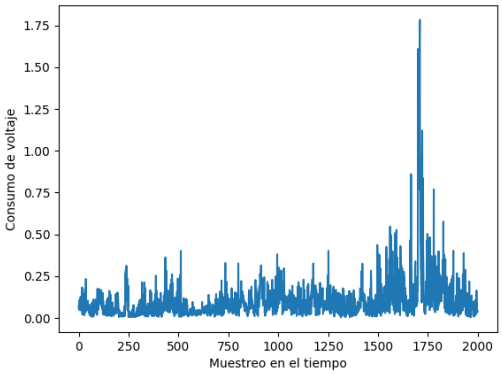
snr

Al igual que en el caso de GIFT-64, SNR para determinar los puntos clave donde las fugas de potencia eran más significativas. Este algoritmo permitió localizar los puntos donde la relación entre los valores intermedios y las trazas de potencia era más fuerte, facilitando la recuperación de las claves de ronda.

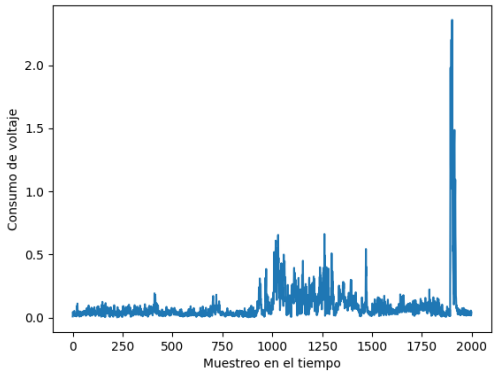
La Figura 5.7 muestra los resultados del análisis SNR para GIFT-128 para las distintas rondas (1-4). Los picos en la gráfica indican los momentos en los que la correlación con la clave supuesta es más alta. El pico principal fue utilizado como punto de referencia para realizar la correlación entre los valores supuestos y las trazas reales, permitiendo extraer la información necesaria para revertir el cálculo de las claves.

5.3. Implementación *fixslicing*

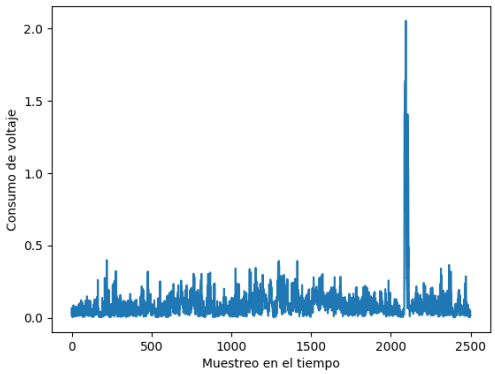
La implementación *fixslicing* se centró exclusivamente en GIFT-128. Este enfoque reorganiza las operaciones internas del cifrador para optimizar el procesamiento y minimizar el costo computacional, lo que resulta en patrones únicos y consistentes en las trazas de potencia. A diferencia de las versiones secuencial y *bitslicing*, *fixslicing*



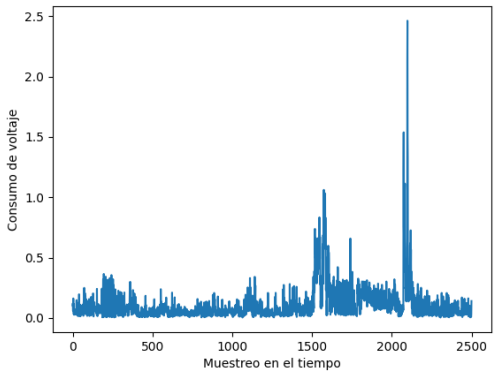
(a) Taza señal a ruido ronda 1



(b) Taza señal a ruido ronda 2



(c) Taza señal a ruido ronda 3



(d) Taza señal a ruido ronda 4

Figura 5.7: Gráficas de señal a ruido de rondas de la versión secuencial GIFT-128 *bitslicing*

genera una distribución interna optimizada que afecta cómo y dónde aparecen las fugas de información.

5.3.1. GIFT-128

El análisis de GIFT-128 se enfocó en las rondas 1,2,3 y 4, que son las primeras donde las claves de ronda interactúan con el estado interno del cifrador. Estas interacciones generan transiciones significativas que pueden observarse en las trazas de potencia capturadas.

La Figura 5.8 muestra una traza representativa de esta implementación, donde se observan patrones que corresponden a las 40 rondas de GIFT-128. En particular, el análisis se centró en las primeras 75000 muestras, que es donde se encuentran las fugas de potencia más relevantes para la recuperación de las claves de ronda.

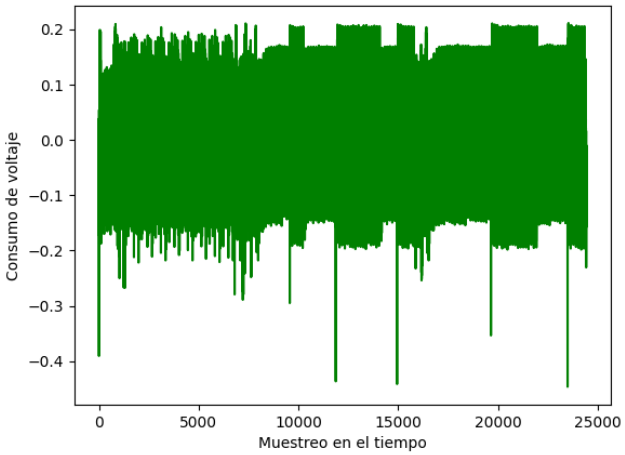


Figura 5.8: Traza capturada de GIFT-128 *fixslicing*.

snr

Con snr se localizaron los puntos críticos donde las fugas de potencia estaban más correlacionadas con los valores intermedios simulados. Los resultados, presentados en la Figura 5.9, destacan los picos que indican las regiones de mayor relación entre las claves supuestas y las mediciones.

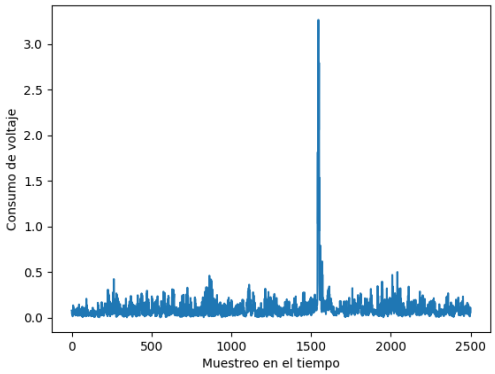
El punto con la mayor amplitud en el análisis marcó la posición clave utilizada para correlacionar los valores simulados con las trazas reales. En esta implementación, el esquema *fixslicing* mostró que estas fugas se concentran en zonas específicas de las trazas, lo que confirma un patrón definido asociado a su estructura optimizada. Es decir que es donde se encuentra la interacción de la clave con las rondas de GIFT-128.

Score

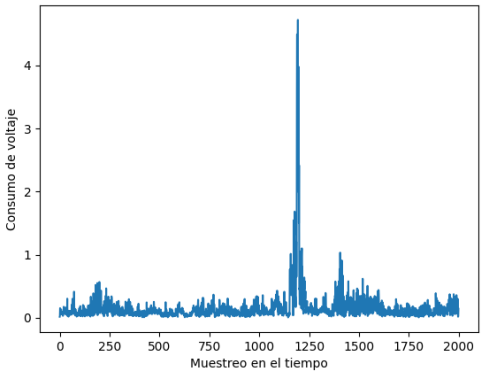
El término `score` se refiere a una métrica que mide la relación entre las trazas capturadas y los valores intermedios simulados para cada posible candidato de clave. En este contexto, un `score` más alto indica que un candidato de clave tiene una mayor probabilidad de ser correcto, ya que su comportamiento se alinea más estrechamente con las fugas de potencia observadas.

La Figura 5.10 ilustra cómo evoluciona el `score` a medida que se incrementa el número de trazas procesadas. Las líneas negras representan el `score` del candidato correcto, mientras que las líneas grises corresponden a candidatos incorrectos. A medida que se utilizan más trazas, se observa cómo el `score` del candidato correcto comienza a destacarse significativamente sobre los demás.

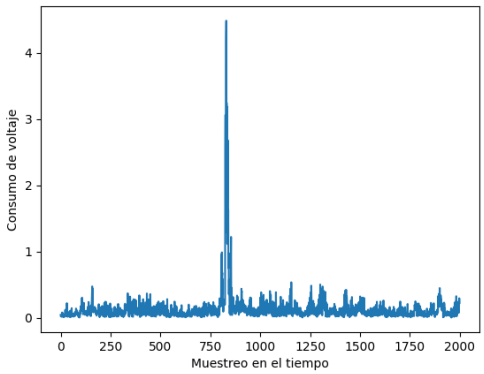
Este comportamiento indica que el ataque fue capaz de identificar el candidato correcto después de procesar un número suficiente de trazas. Este tipo de análisis es clave para evaluar la eficacia del ataque, ya que permite cuantificar el número de trazas necesarias para recuperar la clave de manera confiable.



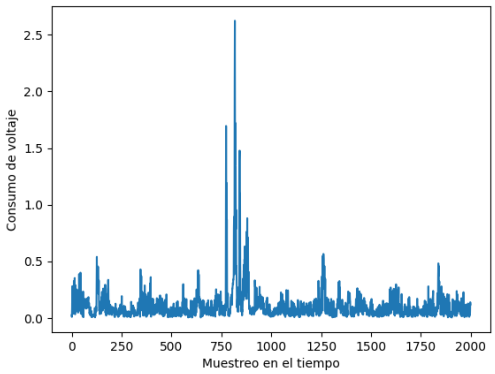
(a) Taza señal a ruido ronda 1



(b) Taza señal a ruido ronda 2



(c) Taza señal a ruido ronda 3



(d) Taza señal a ruido ronda 4

Figura 5.9: Gráficas de señal a ruido de rondas de la versión secuencial GIFT-128 *fixslicing*

5.4. Porcentaje de éxito de las distintas versiones

La Tabla 5.1 resume los resultados obtenidos al realizar experimentos con el cifrador GIFT en diferentes configuraciones, como las implementaciones secuencial, *bitslicing* y *fixslicing*. En estos experimentos, se consideró un ataque exitoso cuando se logró recuperar las claves internas de las rondas del cifrado y reconstruir con éxito la clave completa del algoritmo, lo que representa una medida clave de efectividad en los ataques.

Las implementaciones secuenciales, tanto de GIFT-64 como de GIFT-128, mostraron el mejor desempeño, con tasas de éxito del 99.1 % y el 100 %, respectivamente. Estas pruebas se realizaron con 5000 trazas y emplearon el dispositivo ChipWhisperer Nano. En cambio, las configuraciones basadas en *bitslicing* y *fixslicing* procesaron un mayor volumen de datos, con 50000 trazas cada una, pero sus tasas de éxito fueron

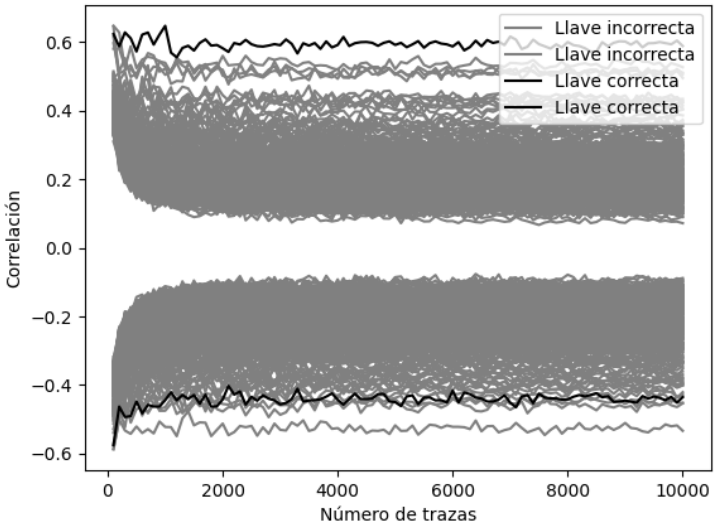


Figura 5.10: Gráfica del score de las posibles claves en GIFT-128 *fixslicing*.

menores, entre el 76 % y el 88 %. Estas configuraciones se probaron en el ChipWhisperer Lite de 32 bits, un dispositivo diseñado para análisis más intensivos en términos computacionales.

Tabla 5.1: Porcentaje de éxito de CPA a las distintas versiones de GIFT-COFB.

Version del cifrador	**Experimentos**	**Éxitos**	**Tasa de éxito (%)**	**Número de trazas**	**ChipWhisperer**
GIFT-64 secuencial	120	119	99.1	5000	Nano
GIFT-128 secuencial	120	120	100	5000	Nano
GIFT-64 *bitslicing*	1000	880	88	50000	Lite
GIFT-128 *bitslicing*	1000	850	85	50000	Lite
GIFT-128 *fixslicing*	5000	4200	76	50000	Lite

Al comparar los resultados obtenidos con el cifrador GIFT en sus diferentes variantes (secuencial, *bitslicing* y *fixslicing*) con los reportados en el estado del arte, se observan algunas tendencias interesantes. Primero, los experimentos realizados con el cifrador GIFT-128 secuencial y GIFT-64 secuencial lograron tasas de éxito muy altas, con un 99.1 % y un 100 % de éxito respectivamente, utilizando únicamente 5000 trazas y un dispositivo ChipWhisperer Nano. Estos resultados coinciden con lo informado por Turan N. [38] para implementaciones con protección de máscara booleana, que también reportan un rendimiento bastante alto con configuraciones similares, aunque en plataformas diferentes. En cuanto a los resultados de GIFT-64 *bitslicing* y GIFT-128 *bitslicing* con los de otras investigaciones, como las de Benjamin A. [39] y Unger W. [40], se observa una ligera disminución en las tasas de éxito. Aunque estos métodos de sin protección han demostrado ser eficientes en ejecución, las tasas de éxito alcanzadas en nuestras pruebas (88 % y 85 % respectivamente, de *bitslicing* que se realizaron en esta tesis) fueron algo menores, lo que podría reflejar una mayor

dificultad para recuperar las claves en estas implementaciones. En contra parte, los resultados con GIFT-128 *fixslicing* mostraron una tasa de éxito del 76 %, lo que, aunque más bajo que los anteriores, sigue siendo un rendimiento notable considerando que este método involucra una mayor complejidad en la implementación.

En resumen, aunque las implementaciones de *bitslicing*, tanto con como sin protección, muestran una disminución en la tasa de éxito en comparación con las versiones secuenciales, continúan siendo métodos efectivos para atacar al cifrador GIFT, especialmente cuando se dispone de un número adecuado de trazas y la herramienta adecuada, como el ChipWhisperer Lite. Estos resultados destacan la importancia del tipo de implementación y la cantidad de trazas en la eficacia del ataque.

Capítulo 6

Conclusiones

En este trabajo se evaluó la seguridad de las implementaciones de software del algoritmo GIFT-COFB frente a ataques por canales laterales, utilizando CPA para identificar posibles fugas de información en las variantes *fixslicing* y *bitslicing*. Los resultados obtenidos revelaron que *fixslicing* proporciona una mayor protección frente a los ataques, comparado con la variante *bitslicing*. Esto se debe a que *fixslicing* introduce más variabilidad en el proceso de cifrado, lo que dificulta la extracción de la clave a través de un análisis de correlación de potencia. Sin embargo, esta protección adicional viene acompañada de una mayor necesidad de trazas para poder realizar una correlación exitosa y recuperar la clave.

Por su implementación de *fixslicing*, esta variante tiende a generar una mayor cantidad de falsos positivos, como se mostró en el análisis del score. Aunque la presencia de estos falsos positivos puede dificultar la identificación precisa del valor correcto de la clave, aun así es posible recuperar la clave con suficientes trazas. Este comportamiento implica que, aunque *fixslicing* es más resistente a los ataques por canales laterales, el costo computacional es significativamente mayor debido a la necesidad de obtener más muestras para realizar una correlación efectiva y lidiar con los falsos positivos.

En cuanto a la cantidad de trazas necesarias para obtener una tasa de éxito aceptable, los experimentos mostraron que las implementaciones *fixslicing* requieren alrededor de 5000 trazas para obtener buenos resultados, mientras que las implementaciones *bitslicing* y las versiones secuenciales permitieron la recuperación de la clave con una menor cantidad de trazas. Esto implica que, si bien *fixslicing* es más seguro frente a los ataques por canales laterales, también presenta un desafío mayor en términos de eficiencia, lo que pone de manifiesto la necesidad de equilibrar seguridad y recursos computacionales.

De manera general, se puede concluir que *fixslicing* es una opción más robusta frente a ataques por canales laterales, pero también más exigente en cuanto a los recursos necesarios para ejecutar el ataque. Este comportamiento pone en evidencia que el modo *fixslicing* no solo es más seguro, sino que también implica una mayor dificultad para los atacantes, aunque no los hace invulnerables. Los resultados obtenidos reflejan la importancia de tomar en cuenta tanto la eficiencia del algoritmo como

su seguridad práctica al ser implementado en software.

Un aspecto importante a destacar es que GIFT-COFB, que fue finalista en la competencia del NIST para algoritmos de cifrado ligero, se muestra vulnerable a los ataques por canales laterales, a pesar de su participación en dicha competencia. Los experimentos realizados en este trabajo demuestran que, independientemente del modo COFB utilizado, el cifrador sigue siendo susceptible a los ataques CPA. Esto subraya la necesidad de diseñar implementaciones más seguras que mitiguen las vulnerabilidades frente a ataques de este tipo, especialmente en aplicaciones críticas que manejan información sensible.

Finalmente, se observa que en el estado del arte hay una escasez de trabajos que aborden específicamente los ataques a GIFT-COFB en sus versiones implementadas en software. La mayoría de los estudios disponibles se centran en versiones secuenciales del algoritmo, con la excepción de un trabajo que propone una protección mediante máscaras booleanas en el modo *bitslicing*. Esta falta de investigación sobre los modos *fixslicing* y *bitslicing* de GIFT-COFB destaca la necesidad urgente de realizar más estudios que analicen la resistencia de estas implementaciones frente a los ataques por canales laterales, dada la relevancia creciente de este cifrador en el campo de la criptografía ligera.

Bibliografía

[1] Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016. [accedido el 7 de Noviembre de 2022].

[2] Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin. Fixslicing: A new gift representation. Cryptology ePrint Archive, Paper 2020/412, 2020. Consultado el 19 de Octubre de 2022.

[3] NewAE. Chipwhisperer-lite, 2024.

[4] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[5] G Joy Persial, M Prabhu, and R Shanmugalakshmi. Side channel attack-survey. *Int. J. Adv. Sci. Res. Rev*, 1(4):54–57, 2011.

[6] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*, pages 104–113. Springer, 1996.

[7] ña practical implementation of the timing attack.

[8] Werner Schindler. A timing attack against RSA with the chinese remainder theorem. In *Cryptographic Hardware and Embedded Systems—CHES 2000: Second International Workshop Worcester, MA, USA, August 17–18, 2000 Proceedings 2*, pages 109–124. Springer, 2000.

[9] Alejandro Hevia and Marcos Kiwi. Strength of two data encryption standard implementations under timing attacks. *ACM Transactions on Information and System Security (TISSEC)*, 2(4):416–437, 1999.

[10] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[11] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology—CRYPTO'97: 17th Annual International Cryptology*

Conference Santa Barbara, California, USA August 17–21, 1997 Proceedings 17, pages 513–525. Springer, 1997.

[12] Ross Anderson and Markus Kuhn. Low cost attacks on tamper resistant devices. In *International Workshop on Security Protocols*, pages 125–136. Springer, 1997.

[13] Sergei P Skorobogatov and Ross J Anderson. Optical fault induction attacks. In *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*, pages 2–12. Springer, 2003.

[14] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 78–92. Springer, 2000.

[15] Roman Novak. SPA-based adaptive chosen-ciphertext attack on RSA implementation. In *International Workshop on Public Key Cryptography*, pages 252–262. Springer, 2002.

[16] Colin D Walter. Sliding windows succumbs to big mac attack. In *Cryptographic Hardware and Embedded Systems—CHES 2001: Third International Workshop Paris, France, May 14–16, 2001 Proceedings 3*, pages 286–299. Springer, 2001.

[17] Faisal Rahman Nuradha, Septafiansyah Dwi Putra, Yusuf Kurniawan, and Muhammad Adli Rizqulloh. Attack on aes encryption microcontroller devices with correlation power analysis. In *2019 International Symposium on Electronics and Smart Devices (ISESD)*, pages 1–4. IEEE, 2019.

[18] Colin O'Flynn and Zhizhang David Chen. Side channel power analysis of an aes-256 bootloader. In *2015 IEEE 28th Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 750–755. IEEE, 2015.

[19] Mostafa Taha and Patrick Schaumont. Side-channel analysis of mac-keccak. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 125–130. IEEE, 2013.

[20] Brian Koziel, Reza Azarderakhsh, and David Jao. Side-channel attacks on quantum-resistant supersingular isogeny diffie-hellman. In *Selected Areas in Cryptography–SAC 2017: 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers 24*, pages 64–81. Springer, 2018.

[21] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 411–436, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[22] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 123–153, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[23] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The simon and speck lightweight block ciphers. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[24] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: A small present. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 321–345, Cham, 2017. Springer International Publishing.

[25] Andrea Caforio, Daniel Collins, Subhadeep Banik, and Francesco Regazzoni. A small gift-cofb: Lightweight bit-serial architectures. Cryptology ePrint Archive, Paper 2022/955, 2022. https://eprint.iacr.org/2022/955.

[26] Seiichi Matsuda and Shiho Moriai. Lightweight cryptography for the cloud: Exploit the power of bitslice implementation. In *Cryptographic Hardware and Embedded Systems–CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings 14*, pages 408–425. Springer, 2012.

[27] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1:5–27, 2011.

[28] Maamar Ouladj. *Side-Channel Analysis of Embedded Systems : An Efficient Algorithmic Approach / by Maamar Ouladj, Sylvain Guilley.* Springer International Publishing, Cham, 2021.

[29] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[30] Eric Brier, Christophe Clavier, and Francis Olivier. Optimal statistical power analysis. Cryptology ePrint Archive, Paper 2003/152, 2003. https://eprint.iacr.org/2003/152.

[31] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis: A generic side-channel distinguisher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442. Springer, 2008.

[32] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, pages 16–29, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[33] François-Xavier Standaert. Introduction to side-channel attacks. *Secure integrated circuits and systems*, pages 27–42, 2010.

[34] Yusuke Yano, Kengo Iokibe, Yoshitaka Toyota, and Toshiaki Teshima. Signal-to-noise ratio measurements of side-channel traces for establishing low-cost countermeasure design. In *2017 Asia-Pacific International symposium on electromagnetic compatibility (APEMC)*, pages 93–95. IEEE, 2017.

[35] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[36] Alex Dewar, Jean-Pierre Thibault, and Colin O'Flynn. Naean0010: Power analysis on fpga implementation of aes using cw305 & chipwhisperer r o, 2020.

[37] NewAE. Chipwhisperer-nano, 2024.

[38] Meltem Sonmez Turan, Meltem Sonmez Turan, Kerry McKay, Donghoon Chang, Lawrence E Bassham, Jinkeon Kang, Noah D Waller, John M Kelsey, and Deukjo Hong. *Status report on the final round of the NIST lightweight cryptography standardization process*. US Department of Commerce, National Institute of Standards and Technology, 2023.

[39] Alexander Benjamin, Jack Herzoff, Liljana Babinkostova, and Edoardo Serra. Deep Learning Based Side Channel Attacks on Lightweight Cryptography (Student Abstract). In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 12911–12912, 2022.

[40] William Unger, Liljana Babinkostova, Mike Borowczak, and Robert Erbes. Side-channel leakage assessment metrics: A case study of gift block ciphers. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 236–241. IEEE, 2021.

Apéndice A

A.1. Instalación de ChipWhisperer

Instalación en Ubuntu 22.04 LTS, para instalar ChipWhisperer y utilizar las libretas de python se realiza con lo siguientes comandos:

• Prerrequisitos

```
sudo apt update && sudo apt upgrade

sudo apt-get install build-essential gdb lcov pkg-config \
    libbz2-dev libffi-dev libgdbm-dev libgdbm-compat-dev liblzma-dev \
    libncurses5-dev libreadline6-dev libsqlite3-dev libssl-dev \
    lzma lzma-dev tk-dev uuid-dev zlib1g-dev curl

sudo apt install libusb-dev make git avr-libc gcc-avr \
    gcc-arm-none-eabi libusb-1.0-0-dev usbutils
```

• Instalación de los ambientes de python y configuración los ambientes virtuales

```
curl https://pyenv.run | bash
    echo 'export PATH="~/.pyenv/bin:$PATH"' >> ~/.bashrc
    echo 'export PATH="~/.pyenv/shims:$PATH"' >> ~/.bashrc
    echo 'eval "$(pyenv init -)"' >> ~/.bashrc
    echo 'eval "$(pyenv virtualenv-init -)"' >> ~/.bashrc

source ~/.bashrc

pyenv install 3.9.5
pyenv virtualenv 3.9.5 cw
pyenv activate cw
```

• Instalación del conjunto de herramientas de ChipWhisperer

```
cd ~/
git clone https://github.com/newaetech/chipwhisperer
cd chipwhisperer
```

• Configuración de las reglas udev para que se puedan reconocer las tarjetas de desarrollo y acceder a ellas mediante USB

```
sudo cp hardware/50-newae.rules /etc/udev/rules.d/50-newae.rules
sudo udevadm control --reload-rules
```

• Creación usuario y grupos que tienen permiso para ChipWhisperer

```
sudo groupadd -f chipwhisperer
sudo usermod -aG chipwhisperer $USER
sudo usermod -aG plugdev $USER
```

• Instalación las librerías para las libretas jupyter

```
git submodule update --init jupyter
python -m pip install -e .
python -m pip install -r jupyter/requirements.txt
cd jupyter
python -m pip install nbstripout
nbstripout --install
```

• Otra alternativa puede instalarse directamente ChipWhisperer como una biblioteca de python3 con el comando pip

```
pip install chipwhisperer
```

A.2. Libretas python y carga del firmware

Chipwhisperer provee de su repositorio con libretas de python para poder realizar los ACL. Estas libretas son utilizadas para el aprendizaje, el curso de SCA y inyección a fallos, como plataforma de experimentación e incluyen las demostraciones del uso de las herramientas chipwhisperer. Las libretas son una plantilla excelente para realizar experimentaciones sobre algún otro algoritmo que no se incluya en chipwhisperer o nuevas formas de analizar los datos de las trazas obtenidas. Dentro del repositorio chipwhisperer se tiene el repositorio `jupyter` que contiene a las libretas. Sin embargo, puede clonarse por separado con el repositorio de chipswhisperer-jupyter:

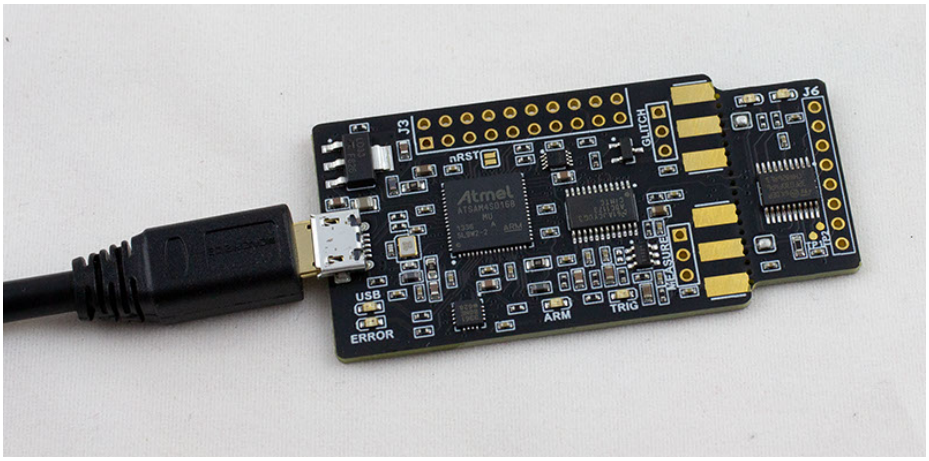


Figura A.1: Conexión del ChipWhisperer Nano

```
git clone https://github.com/newaetech/chipwhisperer-jupyter.git
```

En las libretas python en necesario establecer las siguientes variables globales que van a indicar el tipo de plataforma que se utilizara y el algoritmo estas son:

- `SCOPETYPE` : Indica que hardware de captura se va a utilizar `'OPENADC'` para ChipWhisperer Lite o Pro y `'CWNANO'` para la versión Nano.
- `PLATFORM` : Selecciona el hardware que se esta atacando, `'CW308_STM32F3'` para el microprocesador STM32F3, `'CWLITEXMEGA'` para XMEGA y `'CWNANO'` para el micrprocesador de Chipwhisperer Nano STM32F0.
- `CRYPTO-TARGET` : Indica la biblioteca criptográfica que se va a utilizar. Aquí es donde se establece el algoritmo propio que se va a probar.
- `SS_VER` : Define la versión, utilizar la versión `'SS_VER_1_1'`.

• Ejemplo Utilizando la plataforma ChipWhisperer Nano, con la Versión 1.1, utilizando el algoritmo GIFT64 caja S:

```
SCOPETYPE = 'CWNANO'
PLATFORM = 'CWNANO'
CRYPTO_TARGET = 'GIFT64CSBOX'
SS_VER = 'SS_VER_1_1'
```

Antes de ejecutar alguna celda del código las plataformas deben de estar conectadas mediante un cable micro USB a USB con el equipo que va a procesar y analizar las trazas, tal como lo muestran la figuras A.1 y A.2.

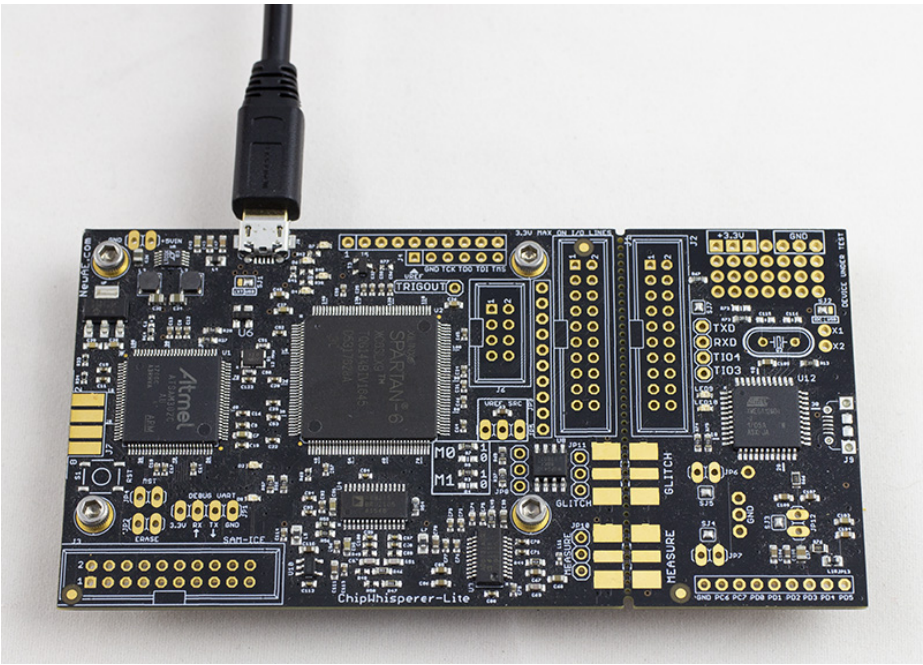


Figura A.2: Conexión del ChipWhisperer Lite

Una vez conectadas las plataformas de ejecuta el siguiente código. Este script va a establecer la conexión con la tarjeta ChipWhisperer, va a establecer el programador para el microcontrolador que se ha configurada y restablecerá el firmware del microcontrolador.

```
%run ../chipwhisperer-jupyter/Setup_Scripts/Setup_Generic.ipynb
```

Si ha encontrado exitosamente el dispositivo, imprimirá el siguiente mensaje:

```
INFO: Found ChipWhisperer
```

Una vez indicadas las variables globales se realiza la compilación cruzada con `arm-none-eabi-gcc`, se ligan las librería y se generan los archivos binario, hex y elf. Genera el firmware de la víctima. (algo) se sustituye por el algoritmo que se va a probar.

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET" "$SS_VER"
cd hardware/victims/firmware/simpleserial-(algo)
make PLATFORM=$1 CRYPTO_TARGET=$2 SS_VER=$3
```

Después de generar el firmware, se carga en la memoria flash de microcontrolador.

(algo) se sustituye por el algoritmo que se va a probar.

```
cw.program_target(scope, prog,
    'hardware/victims/firmware/simpleserial-gift/
    simpleserial-(algo)-{}.hex'.format(PLATFORM))
```

La ejecucción exitosa indica que se ha detectado el microcontrolador ARM, se elimina la memoria en la región `0x8000000` y programa la memoria flash.

Apéndice B

Apendice 2

B.1. Manejo de las herramientas ChipWhisperer

Utilizando las libretas en python se importa ChipWhisperer.

```
import chipwhisperer as cw
```

B.1.1. Scope

`Scope` es uno de los objetos mas importantes es la API. Este se encarga del control de la parte de la tarjeta ChipWhisperer de captura e inducción a fallos. Para utilizarlo se crea el objeto.

```
scope = cw.scope()
```

Configura el osciloscopio para comenzar la captura/inyección de fallos cuando esté activado.

```
scope.arm()
```

Captura las trazas. Debe ejecutarse el método `arm` antes, despues de que se haya capturado una traza, desarma el scope y escribe los datos de regreso. Los datos regresan como un arreglo de NumPy.

```
scope.capture()
```

Desconecta el objeto scope.

```
scope.dis()
```

Establece el numero de muestras a capturar.

```
scope.adc.samples  = num_samples
```

Regresa todas las trazas capturadas en punto flotante. Son valores escalados y recorridos entre $[-0.5, 0.5]$ Dependiendo del tipo de ADC puede regresar 10 bits para la versión Lite o 8 bits para la versión Nano.

```
scope.get_last_trace()
```

B.1.2. Target

Este objeto de la API provee de las interfaces para configurar el dispositivo de prueba (microcontrolador a atacar), para programarlo se utiliza el UART. Para utilizarlo se realizar.

```
import chipwhisperer as cw
scope = cw.scope()
target = cw.target(scope, cw.targets.SimpleSerial)
```

Manda la llave `key` que va a utilizar el algoritmo de cifrado.

```
target.set_key(key)
```

Escribe por el serial al dispositivo objetivo. Al escribir descarta los datos de la memoria `text` y escribe en el buffer. Con el comando `'p'` se inicia la escritura por el serial.

```
target.simpleserial_write('p', text)
```

Lee el serial de dispositivo de prueba. Se recibe la carga útil en un arreglo de bytes codificado en ASCII. Se manda el tamaño del arreglo esperado y el comando `'r'` para leer.

```
response = target.simpleserial_read('r', 16)
```

Desconecta el objeto target.

```
target.dis()
```

B.1.3. Auxiliares

Clase para generar llaves y textos planos básicos. Con el método `next()` se obtiene la siguiente par de llave con el texto plano.

```
import chipwhisperer as cw
ktp = cw.ktp.Basic()
key, text = ktp.next()
```

B.1.4. Código plantilla para realizar la captura de los datos

Una vez establecidas cada una de las clases y objetos con sus métodos explicados, es lo necesario para controlar la tarjeta desde el hardware de captura y el dispositivo de prueba. Después de realizar la configuración descrita en el Apéndice A.1 puede ejecutarse el siguiente código. Con el código se obtienen 2000 conjuntos de trazas. Con cada `scope.arm()` manda la señal y se inicia el cifrado de los datos con un nuevo texto plano, a su vez se van capturando las señales de consumo de potencia. Se genera un arreglo de 2000 conjuntos de trazas para su posterior análisis.

Dentro del análisis de las trazas obtenidas se utilizan las bibliotecas de python para graficar con Matplotlib y Numpy para manipulad los datos como arreglos. Se usan lo métodos estadísticos para buscar las correlaciones, estos incluyen la media, covarianza, desviación estándar, entre otros. Se compara el módelo de consumo de potencia teórico contra los datos obtenidos de las trazas. Posterior al análisis se determina si se ha encontrado la llave con éxito.

```
from tqdm.notebook import trange
import numpy as np
import time

ktp = cw.ktp.Basic()
trace_array = []
textin_array = []

key, text = ktp.next()

target.set_key(key)

N = 2000
for i in trange(N, desc = 'Capturando trazas'):
    scope.arm()

    target.simpleserial_write('p', text)

    ret = scope.capture()
    if ret:
        print('Tiempo agotado.')
        continue

    response = target.simpleserial_read('r', 16)

    trace_array.append(scope.get_last_trace())
    textin_array.append(text)

    key, text = ktp.next()
```

Apéndice C

C.1. Algoritmo criptográfico personalizado

ChipWhisperer tiene varias implementaciones de algoritmos criptográficos, en general del algoritmo AES en distintos modos de operación. Otros algoritmos criptográficos incluyen DES y RSA También incluyen algunos programas para probar la inyección a fallos como la comparación de contraseñas ingresadas y un cargador de arranque. Cuando se desea implementar un algoritmo personalizado para probarlo contra ataques por canal lateral se recomienda que se utilicen como plantilla alguno de las implementaciones de algún algoritmo que se incluyen con ChipWhisperer.

Usando alguno de los algoritmos como plantilla ayuda a identificar las funciones en C que son controladas desde las herramientas en python. Es decir cunado se establece una llave, se manda un texto plano, se inicia el cifrado y se regresa el texto cifrado. El firmware del dispositivo de prueba se encuentra en 4 partes principales dentro del repositorio de chipwhisperer, esta es el proyecto principal, la capa de abstracción de hardware (HAL) que es la interfaz entre el microcontrolador y el software, los archivos crypto y el simpleserial.

C.1.1. Proyecto principal

El proyecto principal es donde se encuentran los archivos del código fuente, por ejemplo unos tienen el nombre `simpleserial-aes.c` que contiene la función principal `main()` y otras funciones para la lectura escritura en el serial del texto plano y texto cifrado, realizar el cifrado, entre otros. La estructura de los archivos puede observarse en la figura C.1. Todo el proyecto principal está ligado a un Makefile que se encarga de la compilación, aquellos con el subfijo `simpleserial`. Estos archivos se encuentran en la siguiente ruta de directorios `chipwhisperer/hardware/victims/firmware/`. El siguiente código muestra el archivo en C principal de GIFT-64, este se encuentra en la ruta:

`chipwhisperer/hardware/victims/firmware/simpleserial-gift64/simpleserial-gift.c`

```
#include "gift-independant.h"
#include "hal.h"
#include "simpleserial.h"
#include <stdint.h>
#include <stdlib.h>

uint8_t get_key(uint8_t * k, uint8_t len)
{
gift_indep_key(k);     // Manda la llave
return 0x00;           // SimpleSerial OK
}

uint8_t get_pt(uint8_t * pt, uint8_t len)
{
trigger_high();        // pone en alto la señal de captura
gift_indep_enc(pt);    // realiza el cifrado
trigger_low();         // baja la señal de captura
simpleserial_put('r', 8, pt);  // regresa el texto cifrado
return 0x00;           // SimpleSerial OK
}

uint8_t reset(uint8_t * x, uint8_t len)
{
return 0x00;           // SimpleSerial OK
}

int main(void)
{
        // configuración del microcontrolador y el serial
platform_init();
init_uart();
trigger_setup();
simpleserial_init();

        // Indica al serial el tipo de paquetes a buscar
        // k para manda llave, p para mandar el texto plano.
        // Ejecuta las funciones cuando los ha recibido.
simpleserial_addcmd('k', 16, get_key);
simpleserial_addcmd('p', 8, get_pt);

        // Busca paquetes y restablece.
simpleserial_addcmd('x', 0, reset);
while(1)
simpleserial_get();
}
```

Estos proyectos se puede identificar procesos claves, uno es la lectura de la llave, los triggers (aquella señal que indica el inicio de la captura de las trazas), el cifrado de los datos y mandar por el serial información. En el caso de código anterior se tiene la función `get_key` que carga la llave al algoritmo GIFT-64 y la función `get_pt` que dentro de la función inicia el cifrado.

C.1.2. Makefile

Dentro de la misma ruta que el proyecto principal se define el Makefile para compilar el proyecto. En este Makefile solo es necesario cambiar las variables que hagan referencia al algoritmo personalizado que se quiere probar. El siguiente código ejemplifica la modificación utilizando GIFT-64.

```
# Nombre de como se va a llamar los archivos compilados
TARGET = simpleserial-gift

# El archivo C del firmware que se quiere probar
SRC += simpleserial-gift.c


EXTRA_OPTS = NO_EXTRA_OPTS
CFLAGS += -D$(EXTRA_OPTS)

# Nombre de los archivos que se encuentran en la carpeta crypto
ifeq ($(CRYPTO_TARGET),)
  ${info No CRYPTO_TARGET passed - defaulting to GIFT64C}
  CRYPTO_TARGET = GIFT64C
endif

${Building for platform ${PLATFORM} with CRYPTO_TARGET=$(CRYPTO_TARGET)}

# Otros archivos requeridos para el build
include ../simpleserial/Makefile.simpleserial

# Ruta del firmware
FIRMWAREPATH = ../.
include $(FIRMWAREPATH)/Makefile.inc
```

Ahora se puede realizar la compilación del firmware indicando la plataforma que se está utilizando, en este caso `'CWNANO'` para el ChipWhisperer Nano.

```
make -j PLATFORM=CWNANO
```

Antes de realizar una nueva compilación, siempre debe eliminar los compilados.

```
make clean
```

Nota: Una vez escrito el archivo Makefile, el proceso de compilación y carga del programa a la memoria flash, al igual que el control de las entradas al algoritmo criptográfico y captura de traza, puede realizarse desde las libretas de python.

C.1.3. Crypto

Otra de los directorios que se modifican para agregar una implementación de algún algoritmo para probar es `crypto`.Esta se encuentra en la ruta:

`chipwhisperer/hardware/victims/firmware/crypto`

Puede visualizarse en la figura C.1. Pueden agregarse el algoritmo como se desee solo tiene que coincidir con el formato explicado en el Apéndice B. Primero de modificar el Makefile general de la carpeta crypto `Makefile.crypto`. Este comando make permite que los archivos donde de define al algoritmo criptográfico se liguen donde se compila en firmware.

```
else ifeq ($(CRYPTO_TARGET),GIFT-FIRMC)
#
# Crypto Target: GIFT64C
# Crypto Options:
#   None
  include $(FIRMWAREPATH)/crypto/Makefile.giftfirmc
```

Se genera un Makefile especifico para el algoritmo que se va a implementar llamado Makefile.gift64c.

```
########
CRYPTO_LIB = gift-64
SRC += gift.c gift-independant.c
CDEFS += -DGIFT64C
VPATH += :$(FIRMWAREPATH)/crypto/$(CRYPTO_LIB)
EXTRAINCDIRS += $(FIRMWAREPATH)/crypto/$(CRYPTO_LIB)
```

Para que todas las versiones implementadas de GIFT tengan el mismo formato, estableció una definición general de todas las funciones utilizadas `gift-independant.h`

```
#include <stdint.h>

#define KEY_LENGTH 16
#define DEFAULT_KEY 0xbd, 0x91, 0x73, 0x1e, 0xb6, 0xbc, 0x27, 0x13,\
0xa1, 0xf9, 0xf6, 0xff, 0xc7, 0x50, 0x44, 0xe7

void gift_indep_init(void);
void gift_indep_key(uint8_t * key);
void gift_indep_enc(uint8_t * pt);
void gift_indep_enc_pretrigger(uint8_t * pt);
void gift_indep_enc_posttrigger(uint8_t * pt);
void gift_indep_mask(uint8_t * m, uint8_t len);
```

Se presenta la implementación de dichas funciones dependiendo de la versión utilizada, solo se muestra para GIFT64C `gift-independant.c`.

```
#include "gift-independant.h"
#include "hal.h"
#include "gift.h"

#if GIFT64C

void gift_indep_init(void) {}
void gift_indep_key(uint8_t * key) {
GIFT64_ECB_indp_setkey(key);}
void gift_indep_enc(uint8_t * pt) {
GIFT64_ECB_indp_crypto(pt);}
void gift_indep_enc_pretrigger(uint8_t * pt) { }
void gift_indep_enc_posttrigger(uint8_t * pt) {}
void gift_indep_mask(uint8_t * m, uint8_t len) {}
```

A continuación se muestra la implementación completa de la versión GIFT64C.

```
#ifndef _GIFT_H
#define _GIFT_H

#include <stdint.h>

#ifndef GIFT_CONST_VAR
//#define GIFT_CONST_VAR
#define GIFT_CONST_VAR
#endif

#define N 64
#define BLOCK_SIZE 8
#define KEYLEN 16
#define RONDAS 28

void GIFT64_ECB_encrypt(uint8_t * input, uint8_t * key, uint8_t * output);
void GIFT64_ECB_decrypt(uint8_t * input, uint8_t * key, uint8_t * output);

void GIFT64_ECB_indp_setkey(uint8_t * key);
void GIFT64_ECB_indp_crypto(uint8_t * input);

#endif
```

```
#include <string.h>

#include "gift.h"

static uint8_t state[8];

static uint8_t RoundKeys[RONDAS][16];

static uint8_t input_save[16];

static uint8_t * Key;

static uint8_t Sbox[16] = {
0x01, 0x0a, 0x04, 0x0c, 0x06, 0x0f, 0x03, 0x09, 0x02, 0x0d, 0x0b,
        0x07, 0x05, 0x00, 0x08, 0x0e
};

static uint8_t BitPermutation[64] = {
0, 17, 34, 51, 48, 1, 18, 35, 32, 49, 2, 19, 16, 33, 50, 3,
4, 21, 38, 55, 52, 5, 22, 39, 36, 53, 6, 23, 20, 37, 54, 7,
8, 25, 42, 59, 56, 9, 26, 43, 40, 57, 10, 27, 24, 41, 58, 11,
12, 29, 46, 63, 60, 13, 30, 47, 44, 61, 14, 31, 28, 45, 62, 15
};

static uint8_t Constants[48] = {
0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3E, 0x3D, 0x3B, 0x37, 0x2F, 0x1E,
```

```
        0x3C, 0x39, 0x33, 0x27, 0x0E, 0x1D, 0x3A, 0x35, 0x2B, 0x16, 0x2C,
        0x18, 0x30, 0x21, 0x02, 0x05, 0x0B, 0x17, 0x2E, 0x1C, 0x38, 0x31,
        0x23, 0x06, 0x0D, 0x1B, 0x36, 0x2D, 0x1A, 0x34, 0x29, 0x12, 0x24,
        0x08, 0x11, 0x22, 0x04
};

static uint8_t Positions[6] = {
//23, 19, 15, 11, 7, 3
3, 7, 11, 15, 19, 23
};

static void BlockCopy(uint8_t * output, const uint8_t * input, uint8_t len) {
uint8_t i;
for(i = 0; i < len; i++) {
output[i] = input[i];
}
}

static void SubCells() {
uint8_t i;
for(i = 0; i < BLOCK_SIZE; i++) {
state[i] = ((Sbox[(state[i] >> 4) & 0x0f]) << 4) | Sbox[state[i] & 0x0f];
}
}

static void PermBits() {
uint8_t temp[8] = {0x00};
uint8_t i, j, k, mov1, mov2;
for(i = 0; i < BLOCK_SIZE; i++) {
for(j = 0; j < 8; j++) {
k = i*8+j;
mov1 = BitPermutation[k]/8;
mov2 = BitPermutation[k]%8;
temp[mov1] = temp[mov1] ^ (((state[i] >> j) & 0x01) << mov2);
}
}
BlockCopy(state, temp, BLOCK_SIZE);
}

static void AddRoundKey() {
uint8_t i, j, k, mov1, mov2;
uint16_t U = ((Key[3] << 8) & 0xff00) | Key[2];
uint16_t V = ((Key[1] << 8) & 0xff00) | Key[0];
uint16_t RK[2] = {V, U};
for(i = 0; i < 16; i++) {
for(j = 0; j < 2; j++) {
k = 4*i+j;
mov1 = k/8;
mov2 = k%8;
state[mov1] = state[mov1] ^ (((RK[j] >> i) & 0x01) << mov2);
}
```

```
}
}

static void RoundConstants(uint8_t ronda) {
uint8_t i;
uint8_t mov1, mov2;
for(i = 0; i < 6; i++) {
mov1 = Positions[i]/8;
mov2 = Positions[i]%8;
state[mov1] = state[mov1] ^ (((Constants[ronda] >> i) & 0x01) << mov2);
}
mov1 = (N-1)/8;
mov2 = (N-1)%8;
state[mov1] = state[mov1] ^ (0x01 << mov2);
}

static void BigEndianState() {
uint8_t temp[BLOCK_SIZE];
uint8_t i;
for(i = 0; i < BLOCK_SIZE; i++) {
temp[BLOCK_SIZE-i-1] = state[i];
}
BlockCopy(state, temp, BLOCK_SIZE);
}

static void BigEndianKey() {
uint8_t temp[KEYLEN];
uint8_t i;
for(i = 0; i < KEYLEN; i++) {
temp[KEYLEN-i-1] = Key[i];
}
BlockCopy(Key, temp, KEYLEN);
}

static void BigEndian(uint8_t * input) {
uint8_t i;
for(i = 0; i < BLOCK_SIZE; i++) {
input[BLOCK_SIZE-i-1] = state[i];
}
}

static void Initialization(uint8_t * text) {
BlockCopy(state, text, BLOCK_SIZE);
BigEndianState();
}

static void gift64_encrypt_ecb(uint8_t * input) {
Initialization(input);
uint8_t i;

Key = input_save;
```

```
//for(i = 0; i < RONDAS; i++) {
for(i = 0; i < 10; i++) {
SubCells();
PermBits();
AddRoundKey();
RoundConstants(i);

Key = RoundKeys[i];
}
}

static void PrecomputeKeys() {
BigEndianKey();

uint8_t i, j;
uint8_t k1_0, k1_1, k0_0, k0_1;
uint8_t * temp = Key;

for(i = 0; i < RONDAS; i++) {
//BlockCopy(RoundKeys[i], Key, KEYLEN);

k1_0 = ((temp[3] & 0x03) << 6) | ((temp[2] >> 2) & 0x3f);
k1_1 = ((temp[2] & 0x03) << 6) | ((temp[3] >> 2) & 0x3f);
k0_0 = ((temp[0] & 0x0f) << 4) | ((temp[1] >> 4) & 0x0f);
k0_1 = ((temp[1] & 0x0f) << 4) | ((temp[0] >> 4) & 0x0f);

for(j = 0; j < KEYLEN-4; j++) {
RoundKeys[i][j] = temp[j+4];
}

RoundKeys[i][j] = k0_0;
RoundKeys[i][j+1] = k0_1;
RoundKeys[i][j+2] = k1_0;
RoundKeys[i][j+3] = k1_1;

temp = RoundKeys[i];
}
}

void GIFT64_ECB_indp_setkey(uint8_t * key) {
Key = key;
PrecomputeKeys();
BlockCopy(input_save, Key, KEYLEN);
}

void GIFT64_ECB_indp_crypto(uint8_t * input) {
gift64_encrypt_ecb(input);
BigEndian(input);
}
```

```
void GIFT64_ECB_encrypt(uint8_t * text, uint8_t * key, uint8_t * output) {
Key = key;
PrecomputeKeys();

gift64_encrypt_ecb(text);
BigEndianState();
BigEndianKey();
BlockCopy(output, state, BLOCK_SIZE);
}

void GIFT64_ECB_decrypt(uint8_t * text, uint8_t * key, uint8_t * output) {

}
```

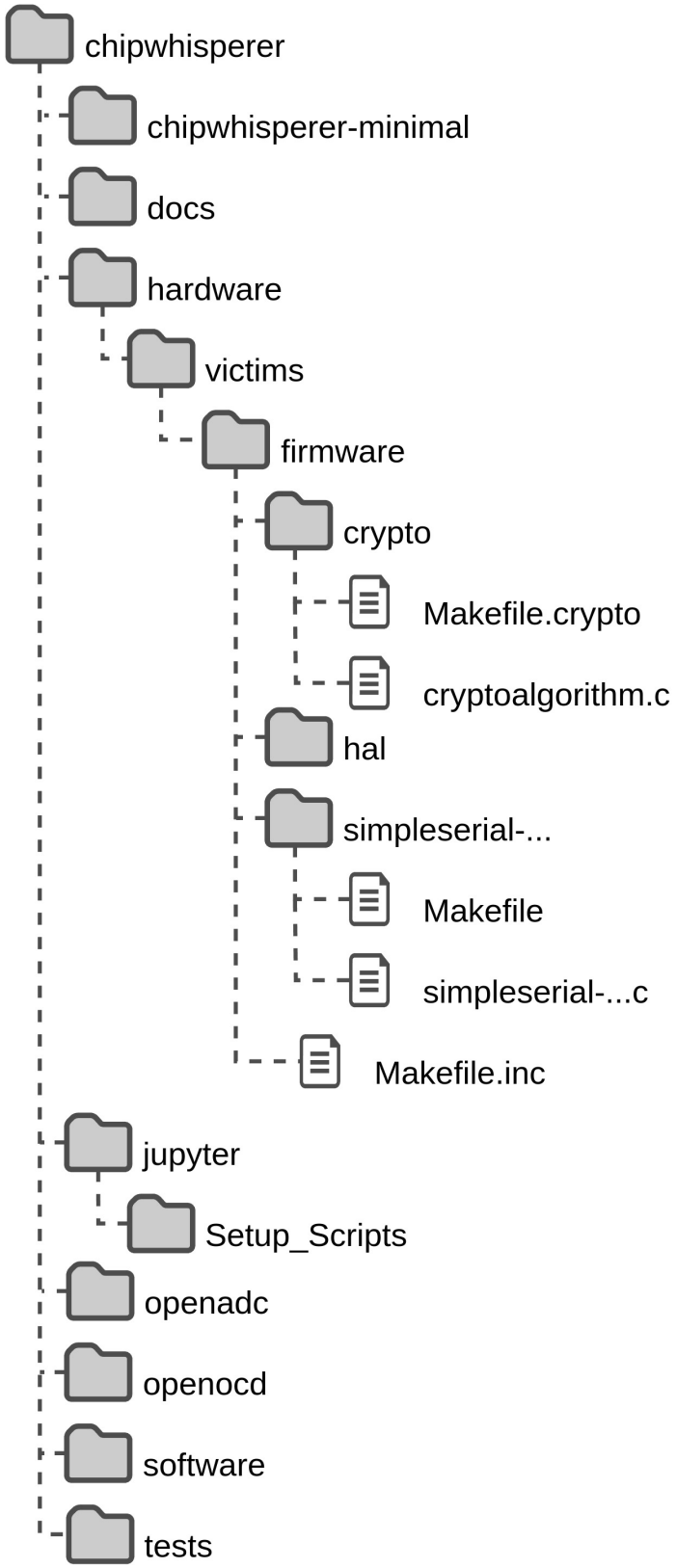


Figura C.1: Sistema de archivos de ChipWhisperer

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará la **Sr. Rogelio Calvillo Juárez**, declaramos que hemos revisado la tesis titulada:

Análisis de la seguridad de implementaciones de GIFT-COFB contra ataques por canales laterales basados en el consumo de potencia

Y consideramos que cumple con los requisitos para obtener el Grado de Maestría en Ciencias en Computación.

Atentamente,

Dr. Cuauhtemoc Mancillas López
Investigador del Departamento de Computación

Dra. Brisbane Ovilla Martínez
Investigador del Departamento de Computación

Dr. Juan Carlos Ku Cauich
Investigador por México

Dra. Sandra Díaz Santiago
Profesora Titular del Departamento de Ciencias e Ingeniería de la Computación de la Escuela Superior de Computo del Instituto Politécnico Nacional