View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Universidad de Zaragoza

Universidad Zaragoza

Trabajo Fin de Grado

Diseño CMOS de funciones físicas no clonables

Autor

Rubén Martín Pinardel

Directores

Santiago Celma Pueyo

Miguel García Bosque

Departamento de Ingeniería Electrónica y Comunicaciones

Junio 2019

Agradecimientos

En primer lugar, me gustaría dar las gracias a mis directores Santiago Celma Pueyo y Miguel García Bosque, cuya disponibilidad y apoyo han sido imprescindibles para la realización de este trabajo. Gracias por las horas dedicadas a la enseñanza y corrección del mismo. También quería dar las gracias a todo el grupo de Diseño Electrónico por la amabilidad y el ambiente excepcional de trabajo. Por último, me gustaría agradecer a Guillermo Díez Señorans su aguante infinito, ya que mi pesadez no ha sido menor.

RESUMEN

Diseño CMOS de funciones físicas no clonables

Este trabajo fin de grado ha tenido como objetivo la introducción teórica a las funciones físicas no clonables (PUF), mediante el conocimiento de sus propiedades y aplicaciones. Además, se ha desarrollado a lo largo del mismo un tipo de estructura concreta de PUF, conformada por osciladores de anillo (RO-PUF).

Para ello, se ha realizado un primer estudio bibliográfico de las PUF, así como de la estructura concreta que íbamos a implementar, considerando ventajas y desventajas. A continuación, se ha introducido el flujo de diseño en la aplicación Vivado, la sintaxis propia del lenguaje de programación hardware Verilog y la arquitectura de la matriz de puertas programables (FPGA). Posteriormente, se han diseñado e implementado varias RO-PUF en FPGA, teniendo en cuenta las peculiaridades y posibles efectos que conlleva dicha implementación.

Finalmente, se han realizado medidas experimentales de la PUF implementada en FPGA y recogido a través de una placa de desarrollo de microcontrolador. Por último, se ha realizado un análisis exhaustivo de las mismas mediante programas de procesado en lenguaje C, con el objetivo final de extraer los parámetros característicos propios de la PUF implementada, así como su adecuación a los parámetros característicos ideales.

Palabras clave: Funciones físicas no clonables (PUF); variabilidad en los procesos de producción; matriz de puertas programables (FPGA); par reto-respuesta (CRP); Vivado; lenguaje de descripción hardware (HDL); inversor; oscilador de anillo (RO); módulos; distancia Hamming (HD); distribución binomial.

Lista de acrónimos

Acrónimo	Significado
RFID	*Radio Frequency Identification*
NVM	*Non-Volatile Memory*
PUF	*Physically Unclonable Function*
RO-PUF	*Ring Oscillator-Physically Unclonable Function*
FPGA	*Field-Programmable Gate Array*
HDL	*Hardware Description Language*
CRP	*Challenge-Response Pair*
ECC	*Error Correction Codes*
RTL	*Register-Transfer Logic*
CLB	*Configurable Logic Block*
LUT	*Look Up Tables*
HD	*Hamming Distance*
FAR	*False Acceptance Rate*
FRR	*False Rejection Rate*
NFA	*Number of False Accepted*
NAA	*Number of Attacker Attempts*
NFR	*Number of False Rejections*
NIA	*Number of Identification Attempts*

Lista de figuras

Figura 1. Esquema conceptual de un oscilador de anillo. 6
Figura 2. Efecto de la variación sistemática en las frecuencias de los osciladores en función de su posición: (a) con variación sistemática y (b) sin variación sistemática. 11
Figura 3. Arquitectura conceptual de la RO-PUF... 12
Figura 4. Submatriz de osciladores con configuración vertical que componen la RO-PUF. 12
Figura 5. Descripción esquemática RTL de la RO-PUF implementada. Compuesta por 12 osciladores de anillo. 15
Figura 6. Efecto de la temperatura en los pares de osciladores 17
Figura 7. Configuración horizontal (izquierda) y vertical (derecha) de un oscilador.................. 18
Figura 8. Gráfica de las medidas experimentales de frecuencia en función del número de inversores obtenidas para la configuración horizontal y vertical. 18
Figura 9. Distancia Hamming intra-chip para diferentes valores del número de cuentas. 20
Figura 10. Posición de las distintas PUF implementadas en la matriz de la FPGA. 21
Figura 11. Distancias Hamming intra-chip e inter-chip de la RO-PUF implementada. 23
Figura 12. Puesto de trabajo y diagrama de bloques. 24
Figura 13. Distribución binomial de las distancias Hamming intra-chip (roja) e inter-chip (azul), así como representación de la distancia Hamming máxima. II

Lista de tablas

Tabla 1. Estados del oscilador de anillo. 7
Tabla 2. Valores de los parámetros característicos para una PUF ideal y real. 16
Tabla 3. Respuestas de la RO-PUF para la configuración horizontal y vertical en función del número de inversores por oscilador. 19
Tabla 4. Respuestas de la RO-PUF para10000 cuentas y para las distintas posiciones $\boldsymbol{M}_{xy}$ *de la matriz donde se ha implementado.* 22
Tabla 5. Máxima distancia Hamming intra-chip de la RO-PUF para las distintas posiciones $\boldsymbol{M}_{xy}$ *de la matriz donde se ha implementado y número de cuentas.* 22
Tabla 6. Respuestas de la RO-PUF implementada en dos FPGAs diferentes. 23

Índice

1 Introducción 1
1.1 Objetivos del trabajo 2
1.2 Descripción del proyecto y metodología 2
1.3 Herramientas utilizadas 3
2 Funciones físicas no clonables 3
3 Oscilador de anillo 6
4 Metodología de implementación 7
4.1 Flujo de diseño en Vivado 7
4.2 Verilog 8
4.3 Arquitectura de la FPGA 8
5 Implementación del oscilador de anillo 8
6 PUF de osciladores de anillo (RO-PUF) 11
6.1 Módulos de osciladores 13
6.2 Módulo selección 13
6.3 Módulo multiplexor 13
6.4 Módulo comparador 14
6.5 Módulo wrapper y final 14
7 Parámetros característicos 15
8 Caracterización experimental 17
8.1 Caracterización del oscilador de anillo 17
8.2 Caracterización de la PUF de osciladores de anillo (RO-PUF) 19
9 Conclusión 24
10 Bibliografía 25
Anexo A. Distribución de Bernoulli y binomial I
Anexo B. Cálculo distancia Hamming máxima. II
Anexo C. Códigos Verilog IV
C.1 Código de dos módulos correspondientes a los osciladores de anillo de la RO-PUF IV
C.2 Código del módulo selección de la RO-PUF V
C.3 Código del módulo multiplexor de la RO-PUF V
C.4 Código del módulo comparador de la RO-PUF VI
C.5 Código del módulo wrapper y final de la RO-PUF VII
Anexo D. Código Arduino NANO VIII

Anexo E. Código del medidor de distancias Hamming X

Anexo F. Códigos de corrección de error (ECC) XIII

F.1 Código Hamming (7,4) XIII

F.2 Repetición XIII

Anexo G. Descripción y planificación de actividades XIV

1 Introducción

El incremento continuo en la capacidad de almacenar, procesar y transmitir información digital está transformando de un modo radical nuestro entorno en un ecosistema de información. El acceso masivo de dispositivos cotidianos a la red (internet de las cosas, IoT) tiene importantes aplicaciones potenciales en ámbitos tan diversos como la logística, la industria o la sanidad. Sin embargo, la naturaleza distribuida de esta tecnología y las severas restricciones en silicio y potencia introducen un nuevo paradigma de vulnerabilidad, en el que la capa física emerge como el eslabón más débil de unos sistemas donde la criptografía clásica resulta excesivamente cara para ser práctica. Debido a la movilidad inherente de los dispositivos, e.g., *smart phones*, el entorno de operación es no confiable y ello aumenta la vulnerabilidad de todo sistema.

La solución actual para proporcionar seguridad a las comunicaciones o de mecanismos de autenticación a estos dispositivos es el uso de claves secretas almacenadas en memorias no volátiles (NVM) y utilizar algoritmos criptográficos tales como firmas digitales o cifrado. Este enfoque es costoso tanto en términos de área de silicio como de consumo de energía. Además, los sistemas de seguridad basados en memoria no volátil son vulnerables a mecanismos de ataque invasivo. La protección contra tales ataques requiere el uso de circuitos activos de detección/prevención de manipulación indebida que para su funcionamiento precisan de energía permanente.

En este contexto surge una innovadora estrategia de seguridad basada en el uso de funciones físicas no clonables (PUF), capaces de explotar las variaciones estocásticas del proceso de fabricación microelectrónica de dispositivos y circuitos para producir secuencias binarias con interés criptográfico: identificación y autenticación unívoca de dispositivos idénticos en diseño y generación de claves [HAN 11].

Las PUF son una solución para la generación y el almacenamiento de claves secretas sin el requisito de costosas NVM y su circuitería de seguridad adicional. Esto es posible, porque en lugar de almacenar claves en la memoria digital, las PUF derivan las claves de las características físicas del circuito integrado (IC). Esta solución es ventajosa en comparación con el almacenamiento digital seguro estándar por varias razones.

Al tratarse de dispositivos altamente difíciles de copiar (clonar) y con una generación impredecible de números binarios, podemos utilizarlos para generar claves y números, que difícilmente pueden ser reproducidos por una modelización del dispositivo. Es decir, debido a sus peculiaridades intrínsecas aleatorias no pueden ser clonados ni física ni matemáticamente, así como tampoco ser atacados mediante ingeniería inversa. Además, debido a que la clave se produce en el propio dispositivo tantas veces como se quiera, no es necesario el uso accesorio de memorias no volátiles (NVM), lo que supone una reducción de los costes en propósitos identificativos, donde es necesario el almacenamiento de la ID, como el caso de los RFID. También supone un incremento de la seguridad, ya que generalmente las claves que se generan mediante algoritmos *software* deben ser almacenadas en memorias NVM, así como transmitidas, siendo estos los puntos de ataque más comunes. Por ello, al no tener necesidad

con las funciones físicas no clonables de guardar la clave ni de transmitirla (en la mayoría de los casos), podemos incrementar sustancialmente la seguridad digital.

Además, el *hardware* PUF usa circuitos digitales simples que son fáciles de fabricar y consumen menos energía y área que las soluciones basadas en NVM con sus necesarios circuitos de seguridad.

A pesar de las ventajas claras que estos dispositivos pueden llegar a presentar y que han supuesto la participación de una gran cantidad de investigadores en el desarrollo de este tipo de tecnologías, cabe señalar que tiene algunos inconvenientes. Principalmente debido a los errores aleatorios y deterministas que sufren los datos generados. Los errores aleatorios consisten en variaciones en los bits por ruido estadístico, y los errores deterministas se deben a desviaciones sistemáticas, producidas por efectos de cambios de temperatura, variaciones en la tensión de alimentación o degradación del circuito. Por ello, las funciones físicas no clonables precisan un posterior tratamiento de errores, que en casos como el cifrado de mensajes son esenciales y en propósitos de identificación no tanto, ya que estas variaciones de bits son lo suficientemente pequeñas para no suponer un error importante en la identificación.

1.1 Objetivos del trabajo

Los objetivos de este Trabajo Fin de Grado son la introducción teórica al mundo de las funciones físicas no clonables (PUF), desde una contextualización global que nos permita identificar sus propiedades y aplicaciones hasta el análisis y desarrollo específico de estructuras concretas. A partir de este análisis se realizará el diseño específico de PUF conformadas por osciladores de anillo (RO-PUF) y se implementarán en una matriz de puertas programables (FPGA). Finalmente, se realizarán medidas de PUF implementadas en una FPGA para analizar el comportamiento de la misma bajo diferentes parámetros de implementación, así como para la comprobación de la bondad de la PUF bajo determinados criterios teóricos.

1.2 Descripción del proyecto y metodología

El proceso realizado a lo largo del trabajo comprende el análisis teórico, diseño, implementación y posterior caracterización de un circuito de función física no clonable mediante la tecnología de matriz de puertas programables.

La realización de este trabajo ha tenido como esquema principal, el siguiente flujo de trabajo:

- Estudio bibliográfico de las PUF.
- Selección y caracterización teórica tanto del elemento básico constitutivo de la PUF elegida, el oscilador de anillo, como de la RO-PUF.
- Aprendizaje del entorno *software* proporcionado por la herramienta Vivado, así como el aprendizaje del lenguaje de descripción hardware Verilog.
- Estudio de los fundamentos de sistemas digitales y lógica programable.
- Estudio de la arquitectura de la FPGA sobre la que se implementa la PUF.
- Diseño en Verilog del oscilador de anillo.
- Diseño en Verilog de la PUF de osciladores de anillo.

- Caracterización experimental mediante una placa Arduino NANO y posterior análisis de los datos.

La principal dificultad del trabajo ha radicado en la adquisición de conceptos y utilización de herramientas que, a pesar de su amplio uso en el ámbito de la electrónica, no se han impartido a lo largo de la carrera. Aunque sí que se han utilizado conceptos básicos adquiridos en las asignaturas del ámbito de la Electrónica del Grado de Física: Técnicas Físicas I y II, y Electrónica Física.

1.3 Herramientas utilizadas

- **Verilog:** se trata de un lenguaje de descripción hardware (HDL) que se utiliza para modelar sistemas electrónicos. A pesar de tener una sintaxis similar al lenguaje C, este no se ejecuta de forma estrictamente lineal, y basa su diseño en la jerarquía de módulos. Estos módulos son las unidades básicas, y están conformados por una serie de puertos de entrada y salida, así como por dos tipos principales de variables: tipo *wire* y tipo *register*.
- **Vivado:** es una plataforma software producida por Xilinx que se utiliza para la síntesis y análisis de diseños en lenguaje de descripción hardware (HDL).
- **ZyboZynq 7000:** placa de desarrollo de la FPGA de Xilinx "xc7z010clg400-1". Como puerto de entrada utilizaremos un puerto micro USB y las salidas las asignaremos a los puertos PMOD o a LEDs.
- **Arduino NANO:** placa producida por Arduino que dispone entre otros dispositivos de un microcontrolador (ATmega328) y diversos puertos entrada-salida.

2 Funciones físicas no clonables

Con el objetivo final de entender y desarrollar una función física no clonable, vamos a proporcionar un marco teórico introductorio. Las preguntas clave que nos pueden ayudar en esta contextualización son tres: ¿qué son las PUF?, ¿para qué sirven? y ¿por qué sirven para eso?

En primer lugar, debemos saber qué son las PUF, acrónimo inglés de *Physical Unclonable Functions (*PUF), que se puede traducir al español como funciones físicas no clonables. Tal y como su propio nombre indica, se trata de una serie de dispositivos físicos, teóricamente no copiables y que se comportan como una función matemática, por lo menos en la aproximación teórica. Es decir, se tratan de una serie de dispositivos físicos que para cada entrada (*input*) que introduzcamos, generan una salida (*output*), de manera biyectiva.

Estos dispositivos utilizan la variabilidad inducida en los procesos de fabricación de los dispositivos físicos para generar las distintas respuestas. Esta variabilidad, en general, no es creada a propósito con el fin de generar un comportamiento determinado, sino que se produce por la aleatoriedad estadística propia del proceso de fabricación, de manera que, aunque intentemos realizar dos dispositivos iguales, estos tendrán una serie de diferencias físicas que afectarán a su comportamiento (por eso se denominan inclonables). La señal de entrada se conoce como señal reto (*challenge*) y puede afectar en la manera en la que se combinan las variaciones locales; así como también cuáles se combinan, provocando una

variación en la señal de salida o respuesta (*response*). Cabe destacar que en lo que sigue de trabajo, y como referencia a la gran mayoría de bibliografía escrita sobre PUFs, denotaremos a la entrada de la PUF como reto y a la salida como respuesta, y hablaremos genéricamente de pares reto-respuesta (CRP) [BÖM 13].

A pesar del nombre, algunas PUF han sido en la práctica modelizadas matemáticamente, de manera que se han podido reproducir las CRPs de la misma. Aunque teóricamente no se trata de un clon, ya que no es el mismo dispositivo, sí que tiene el mismo comportamiento, por lo que es considerado un clon. Anteriormente se ha mencionado, que las PUF son funciones matemáticas biyectivas, aunque en la práctica no lo son, ya que todas ellas sufren errores. Por ejemplo, en PUF digitales los distintos bits que conforman la salida se ven alterados aleatoriamente durante su tratamiento debido a ruido, interferencias, distorsión, etc. Por lo que para cada reto puede haber varias respuestas. Más adelante, veremos cómo se tratan los inconvenientes anteriores.

Una vez visto qué son las PUF, vamos a intentar para qué sirven y por qué sirven para eso. Dentro de sus aplicaciones, se distinguen tres grandes campos que son la identificación, autenticación y generación de claves criptográficas [BÖM 13, DEV 11]. Las dos primeras, suelen estar relacionadas, aunque como veremos, no son lo mismo.

Identificación. En el contexto de la microelectrónica y las PUF, la identificación (ID) es la asignación de un código digital a un chip. El requisito básico para la identificación es que para cada dispositivo se asigne una ID única. Esto es importante para evitar identificaciones falsas. Por lo tanto, el proceso de identificación tiene que ser una función inyectiva.

Por ejemplo, las PUF se presentan como dispositivos capaces de reducir costes en la identificación por radio frecuencia (RFID). Al poder generar la identificación cuantas veces se quiera y sin necesidad de memoria donde guardar la misma (ya que esta ID es inherente al PUF y reproducible) se utilizan combinados con los RFID. De manera que estas PUF-RFID, generan una identificación (ID) que se asigna al objeto a identificar y una información en una base de datos. Es por eso que la identificación se puede utilizar para obtener más información sobre la identidad reclamada.

Autenticación. La identificación no significa que la ID reclamada se haya verificado. En general, la identificación no incluye la verificación de identidad. No se asegura que la ID reclamada sea la correcta. De forma que la autenticación consiste en verificar la identidad reclamada.

Para ello, se utiliza el sistema CRP, basado en la generación de pares reto-respuesta. De forma que generamos muchas respuestas procedentes de diferentes retos, las cuales guardaremos también en la base de datos. Tras la identificación del chip se procede a la autenticación, mediante el envío de un nuevo reto al PUF y la comprobación de la respuesta. Sí esta respuesta es acorde a un par *challenge-response* registrado, se da por autenticado. Hay diferentes formas de implementar este sistema, aunque la idea es la misma. Por ejemplo, se puede realizar una generación de CRP dinámica, basada en que solo hay un CRP guardado cada vez en la base de datos, y en cada proceso de autenticación es generado un nuevo CRP. Esto se puede implementar, enviando dos retos que generan dos respuestas, una es utilizada para la

autenticación, y la otra guardada como el nuevo CRP. También se puede combinar con sistemas criptográficos para codificar el envío de estos pares.

Generación de claves. Las respuestas obtenidas por una PUF pueden ser usadas como claves criptográficas, gracias a su imprevisibilidad. Además, debido a que generan por sí mismas la clave (es una respuesta de la PUF) y a que cuando está apagada la PUF, la clave ni siquiera se guarda en memoria; ya que la genera el propio dispositivo, se puede aumentar la seguridad de los sistemas criptográficos.

En la generación de claves criptográficas no podemos hablar de las PUF sin la necesidad de un tratamiento de errores adecuado. Debido a que la clave debe ser siempre la misma. Es por ello que se han desarrollado numerosos *Error Correction Codes* (ECC), algunos de ellos están explicados en el Anexo F.

Con el fin de concretar y definir más precisamente las PUF, se pueden extraer una serie de propiedades inherentes a todas ellas [MAE 12]:

- **Constructibilidad:** se trata de la capacidad real de llevar a cabo el desarrollo de la misma. Mientras que diseñar una sin requerimiento previo suele ser sencillo, en ocasiones nos puede interesar que los CRPs sean de una determinada forma, lo que puede acarrear dificultades.
- **Evaluabilidad:** solo es posible si es construible, y se trata de la capacidad de evaluar la respuesta en función de la entrada y de la PUF.
- **Reproducibilidad:** solo es posible si es evaluable. De manera que para el mismo dispositivo las salidas aun mismo reto serán cercanas entre sí.
- **Exclusividad:** solo es posible si es evaluable. De forma que para dos dispositivos distintos obtenemos dos respuestas distintas para el mismo reto.
- **Identificabilidad:** solo es posible si es reproducible y exclusivo en tal grado que implica que será posible identificar un mismo dispositivo, sin posibles falsas identificaciones con otros dispositivos.
- **Inclonabilidad física:** si es evaluable por otro usuario no autorizado y aun así no es capaz de reproducir el proceso de creación de forma exacta.

El ámbito en el que enmarcaremos principalmente el desarrollo del trabajo es el de la identificación, ya que la generación de claves criptográficas conllevaría un riguroso tratamiento de post procesado mediante códigos de corrección de error. Además, la PUF de osciladores de anillo que vamos a implementar se trata de una PUF débil, mientras que las PUF utilizadas en criptográfica se tratan de PUF fuertes. La diferencia entre ambas es el número de CRPs, las fuertes tienen un número muy elevado (usualmente entre varios cientos y unos pocos millones de CRPs [RUH 14]) que puede llegar a divergir, lo que impide a algoritmos de *machine learning,* entre otros, modelizarlas [HER 14, HAN 11]. Por otro lado, las débiles tienen pocos pares CRP [MAE 12]. Esta propiedad es fundamental, debido a que, si un posible atacante tuviera acceso a la PUF, mediante el análisis de los CRP podría modelar la PUF. Este hecho se evita con las PUF fuertes.

3 Oscilador de anillo

Una vez puesto en contexto este trabajo, vamos a particularizar nuestro caso de estudio. El objetivo final tal y como ya se ha comentado, es implementar una PUF en una FPGA, consistente en una serie de osciladores de anillo. Por lo tanto, en primer lugar, debemos introducir lo que es un oscilador de anillo [RAB 03], así como la forma en que lo vamos a implementar. Un oscilador de anillo no es más que una sucesión impar de inversores lógicos con realimentación de la salida del último a la entrada del primero. Estos inversores tienen un tiempo de propagación entre la entrada y la salida, cuyo promedio denominaremos retardo.

La frecuencia de oscilación de un oscilador de anillo con inversores en una aproximación lineal tiene la siguiente expresión:

$$f = \frac{1}{2N_{inv} \cdot t_p} \qquad (1)$$

Donde N_{inv} es el número de inversores y t_p es el tiempo promedio de propagación e igual a $\frac{(t_{pLH}+t_{pHL})}{2}$. El valor de t_{pLH} es el tiempo de propagación entre los valores bajo y alto de la señal, mientras que t_{pHL} es el tiempo de propagación entre los valores alto y bajo de la señal. Es decir, desde que la entrada cambia de 1 a 0 hasta que la salida cambia de 0 a 1 para t_{pLH} y al revés para t_{pHL}.

El factor 2 es debido a que un periodo completo requiere de ambas transiciones. Esta ecuación solo es válida si $t_p \gg t_r + t_f$, donde t_r y t_f son los tiempos de los flancos ascendente y descendente de la señal. Es por ello que hay un límite inferior de inversores, que suele estar en 5 para la mayoría de tecnologías. La fórmula representa una aproximación para una visualización más sencilla del comportamiento de este oscilador, aunque realmente y es la clave de este trabajo, los retardos de cada inversor serán distintos, lo que nos permitirá implementar la PUF.

Para conseguir dicha señal oscilante, es necesario generar un sistema astable. Esto se consigue como ya hemos comentado mediante una sucesión impar de inversores realimentados, de forma que la salida varié entre los valores "0" y "1" lógicos. Con el fin de obtener un control sobre el funcionamiento del mismo, vamos a añadirle una puerta AND, que utilizaremos como *enable* o habilitador y que nos permitirá controlar la estabilidad del sistema. En la figura 1 se puede observar el esquema conceptual del oscilador y en la tabla 1 se presenta la tabla de verdad del sistema anterior:

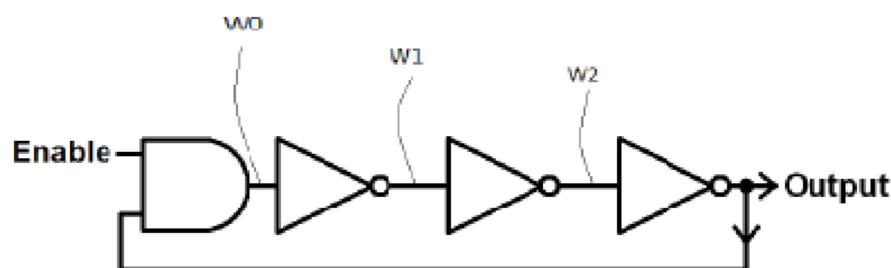

Figura 1. Esquema conceptual de un oscilador de anillo.

Estado	Enable	W_0	W_1	W_2	Output	Condición	
1)	0	0	1	0	1	Estable	Sin oscilación
2)	1	0	1	0	1	Inestable (decae a estado 3)	Oscilación
3)	1	1	0	1	0	Inestable (decae a estado 2)	

Tabla 1. Estados del oscilador de anillo.

En la tabla anterior podemos observar cómo cuando por la entrada *enable* tenemos un cero (estado 1), el sistema es estable ya que W_0 va ser siempre cero. En cambio cuando introducimos un "1" lógico por el *enable* (estado 2 y 3), W_0 va cambiar en función de si el output es "0" o "1", de la misma forma que el output va a cambiar en función del valor de W_0, generándose así la oscilación.

4 Metodología de implementación

4.1 Flujo de diseño en Vivado

Una vez entendido el comportamiento de un oscilador de anillo debemos de implementarlo en la FPGA. Para ello, es necesario seguir un flujo de diseño [NVT 13] que consta de los siguientes pasos:

1. Programación: en lenguaje Verilog de todos los módulos, variables y uniones de los que va a constar nuestro diseño.

2. Elaboración: el archivo en Verilog que hemos programado se traduce en lo que se denomina descripción *Register-Transfer Logic* (RTL), que no es más que una descripción equivalente mediante elementos lógicos ideales de nuestro diseño.

3. Síntesis: consiste en sustituir los elementos de nuestro circuito por elementos lógicos reales que se programan en los distintos *Configurable Logic Block* (CLB).

4. Implementación: se realiza ya una descripción física de los elementos lógicos reales sobre la matriz, es decir, cada módulo, conexión, etc., que hayamos programado se coloca en la FPGA.

5. Generación de *Bitstream*: en este paso todo el diseño está terminado y se debe transcribir correctamente a bits para que la FPGA se programe como nosotros hemos diseñado.

Todo el proceso se puede seguir a través de la interfaz que nos proporciona Vivado. Además, es necesaria la inclusión de lo que se conoce como *Constraints*, que son una serie de restricciones físicas que se le pueden aplicar a la FPGA, consistentes principalmente en la configuración de los puertos físicos tanto de entrada como de salida. Vivado también permite

la simulación del circuito, para lo que es necesario la introducción de las condiciones iníciales deseadas. La programación se realiza mediante Verilog, por ello vamos a describir los aspectos básicos del mismo sucintamente.

4.2 Verilog

Verilog basa su diseño en la jerarquía de módulos [STFC 16]. Estos módulos son las unidades básicas, y están conformados por una serie de puertos de entrada y salida, así como por dos tipos principales de variables: tipo *wire* y tipo *register*. Ambas variables toman valores binarios (cero o uno), pero en general el tipo *wire* se actualiza en tiempo real, y el tipo *register* lo hace secuencialmente cuando cambia el reloj. Dentro de estos módulos se pueden definir sentencias que fijan el comportamiento del módulo, de forma similar a C, aunque la ejecución de las mismas se puede realizar de dos modos: asignación continua que solo se puede usar para variables tipo *wire* o asignación procedural que puede ser *blocking* si bloquea la ejecución del código hasta que la ejecución de la sentencia se completa o *non-blocking* si se ejecuta en paralelo con una señal de reloj. Además, se suele hablar de cambios de estado del módulo síncronos o asíncronos en función de si se realiza el cambio por la variación de una señal de reloj o por la variación de otra variable de estado, respectivamente. El cambio se realiza mediante la función *always @()* (dentro del paréntesis va la variable de cambio) y se produce por las asignaciones procedurales que se encuentran dentro del *always*.

Antes de entrar en cómo se haría el oscilador de anillo, vamos a describir sucintamente cómo es la FPGA donde vamos a implementar el diseño.

4.3 Arquitectura de la FPGA

Las FPGAs [MAX 04] están compuestas por los que se conocen como *Configurable Logic Block* (CLB). Estos bloques están dispuestos en forma de matriz ocupando la mayor parte de la FPGA. Se distribuyen en cuatro zonas diferenciadas. En el caso concreto de la FPGA utilizada, “xc7z010clg400-1”, la matriz de CLBs toma unas coordenadas espaciales (M_{xy}) que van de: $0 \leq x \leq 43$ a $0 \leq x \leq 99$. Estas pequeñas unidades están compuestas por dos celdas independientes, conformadas a su vez cada una por cuatro *Look Up Tables* (LUT), las cuales constan de seis entradas y ocho elementos de almacenamiento. Estas LUT básicamente son memorias programables que almacenan una función lógica (tabla de verdad), de forma que en función de las entradas y de cómo las hayamos programado vamos a obtener una determinada salida.

5 Implementación del oscilador de anillo

Como ya hemos mostrado con anterioridad, para construir nuestro oscilador de anillo con capacidad para controlar su estabilidad únicamente vamos a necesitar emular dos tipos de puertas lógicas: la puerta AND y la puerta NOT. El programa Vivado con el que vamos a realizar toda la implementación de la PUF, nos permite programar una a una todas las LUTs de

la FPGA, pudiendo especificar su posición dentro de la misma. Para implementar las puertas lógicas que necesitamos utilizaremos los denominados LUT1 para los inversores; estos disponen únicamente de una entrada, y los LUT2 para la puerta AND; con dos entradas. A partir de un parámetro dispuesto en la cabecera del módulo[1], es posible hacer que estos dos módulos se comporten como nuestras puertas. A continuación, se muestra un ejemplo de sintaxis de dichos módulos:

```
LUT2 #(4'b1000) AND(.O(w[0]), .I0(enable), .I1(out_ro));
LUT1 #(2'b01) inv_0(.O(w[1]), .I0(w[0]));
```

Una vez conocida la implementación de estas puertas, simplemente debemos unirlas y generar el oscilador de anillo anteriormente descrito. El software de Vivado reconoce como un error los bucles combinacionales como el que estamos realizando al implementar el oscilador de anillo, por lo que hay que especificarle la existencia del mismo:

```
(* ALLOW_COMBINATORIAL_LOOPS = "true", KEEP = "true" *) wire out_ro;
```

Una de las partes más importantes consiste en fijar la ubicación de cada uno de los inversores a utilizar. Como ya hemos comentado Vivado permite ubicar concretamente cada uno de estos inversores, para ello es necesario en primer lugar fijar la localización de la CLB dentro de la FPGA y a continuación fijar la LUT en concreto dentro de la CLB que vamos a programar como inversor o puerta AND. Además, es necesaria una condición extra que evita la eliminación por parte de Vivado de los módulos que considera redundantes, como sería nuestro caso al implementar muchos inversores iguales:

```
(* BEL="A6LUT", LOC="SLICE_X0Y0", DONT_TOUCH="true" *)
```

El código completo del oscilador de anillo en Verilog se puede consultar en el Anexo C.1. Tras realizar la programación del oscilador de anillo en Vivado, es necesario ver si la implementación sobre la FPGA se produce de forma satisfactoria. Para ello es necesario comprobar si genera la señal oscilante, así como la variación en frecuencia que sufre en función del número de inversores implementados. Además, queda observar un hecho fundamental para el desarrollo posterior de la PUF, este es la variación sistemática producida en la frecuencia debida a la disposición espacial de los inversores que conforman el oscilador dentro de la matriz. Esta variación sistemática se produce por varios factores entre los que se encuentran principalmente la distinta longitud de conexión entre LUTs en función de su posición en la matriz, la falta de simetría entre osciladores, así como pequeñas variaciones en las condiciones de funcionamiento (temperatura, etc.) en función de la localización en la FPGA.

Si tratásemos de modelizar teóricamente el retraso [MAI 10] que tiene uno de estos osciladores de anillo implementados en la matriz, tendríamos que tener en cuenta tres factores. El primero sería el retraso debido a la suma de retrasos que tendrían todos los

[1] Estos módulos se encuentran ya programados en las bibliotecas que el programa Vivado dispone. Por lo que simplemente es necesario nombrar dicho módulo e introducir un parámetro para especificar su comportamiento.

osciladores de anillo si fuesen implementados con inversores completamente iguales (d_{medio}), el cual es el valor empleado en la formula (1), donde $d_{medio} = 2N_{inv} \cdot t_p$. El segundo retraso a tener en cuenta es la suma de las variaciones aleatorias en el retraso de cada uno de los distintos inversores y uniones que componen el oscilador de anillo ($d_{aleatorio}$), producidas en el proceso de fabricación. Hasta ahora los dos retrasos toman como completamente simétricos los distintos osciladores, es decir, que sus inversores y uniones son simétricas entre sí. Por último, tendríamos que tener en cuenta el retraso introducido por la variación sistemática comentada ($d_{sistemático}$). Por lo tanto, el retardo total del oscilador (o periodo de oscilación) viene dado por la suma de las tres contribuciones anteriores:

$$d_{osc} = d_{medio} + d_{aleatorio} + d_{sistemático} \quad (2)$$

Esta d_{osc} es el retraso total del oscilador de anillo implementado en la FPGA. Tanto $d_{aleatorio}$ como $d_{sistemático}$ pueden ser positivos o negativos, ya que son sumas de variaciones locales. La variación sistemática realmente es un problema a evitar, ya que la PUF se tiene que basar únicamente en la variación aleatoria introducida en el proceso de fabricación. La pregunta que surge ahora es ¿por qué es necesario evitar esta variación sistemática? Para explicarlo, hay que conocer cómo va a funcionar nuestra PUF, y a pesar de que se explicará más adelante, hay que tener en cuenta que los bits de la respuesta se generan al comparar por pares (mediante contadores seguidos de un comparador digital) las frecuencias de múltiples osciladores de anillo implementados en la FPGA. Por tanto, para un par de osciladores de anillo, la diferente frecuencia de oscilación dependerá de la diferencia de retardos dada por:

$$\Delta d_{osc} = (d_{medio} + d_{aleatorio} + d_{sistemático})_1 - (d_{medio} + d_{aleatorio} + d_{sistemático})_2 =$$

$$= \Delta d_{aleatorio} + \Delta d_{sistemático} \quad (3)$$

Eliminándose el retraso común y apareciendo únicamente la variación entre los retrasos introducidos por la variación aleatoria y sistemática como los efectos que producen la variación entre las frecuencias de los osciladores y que por tanto generarán la respuesta.

Si solo tuviéramos en cuenta como efecto de variación, la variación aleatoria, los bits individuales de nuestra salida se ajustarían a la distribución de Bernoulli (Anexo A), es decir, la probabilidad de que saliese "1" o "0" sería igual de probable debido a que suponemos que el resultado es verdaderamente aleatorio, sin sesgo. En términos de dicha distribución, la afirmación anterior se podría exponer como que el valor de la probabilidad de que salga un "1", p, es un medio, al igual que la probabilidad de que salga "0" sería un medio. Sin embargo, las desviaciones sistemáticas, tales como diferencias en la topología o la ubicación espacial de los osciladores, afectan a esta equiprobabilidad. Este resultado puede expresarse de la siguiente manera:

$$p = 0{,}5 \quad si \;\; \Delta d_{sistemático} = 0$$

$$p = 0{,}5 + \Delta p \quad si \;\; \Delta d_{sistemático} \neq 0$$

Por ejemplo, en la figura 2 se representan el efecto sobre la frecuencia de salida de idénticos osciladores de anillo (RO) ubicados en coordenadas diferentes donde se aprecia una variación sistemática en el caso (a) y como sería el caso ideal que buscamos sin variación sistemática (b).

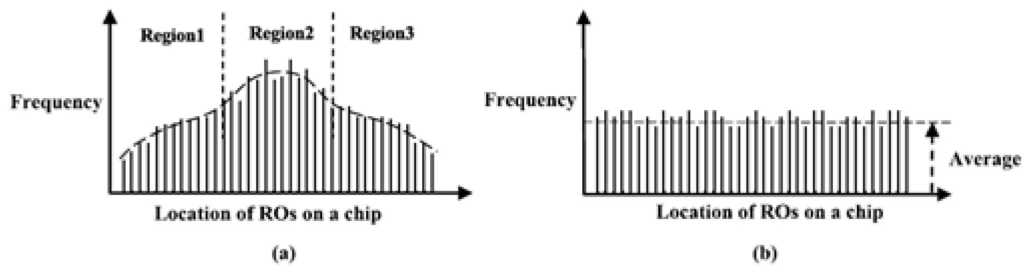

Figura 2. Efecto de la variación sistemática en las frecuencias de los osciladores en función de su posición: (a) con variación sistemática y (b) sin variación sistemática. [MAI 10]

Vemos claramente que si comparamos osciladores de la región dos (Figura 2 (a)) con cualquiera de las otras dos regiones de la FPGA vamos a obtener siempre "1" porque tienen frecuencias mayores. El problema de esta variación sistemática es que, si fabricamos un mismo lote de FPGAs, todas estas van a tener en general una variación sistemática similar debido a los factores de mismas uniones entre LUTs, etc. Por tanto, si implementásemos la misma PUF (misma disposición de osciladores) en diferentes matrices y obtuviésemos las respuestas que nos generan cada una, por este efecto serían bastante similares. Es evidente que afecta seriamente a la exclusividad, es decir, dos chips distintos nos pueden dar para el mismo reto una respuesta muy similar o igual.

Una posible solución a este problema, la cual vamos a implementar en nuestra PUF, consiste en una disposición de osciladores simétrica, así como situar los mismos en posiciones lo más cercanas posibles entre sí. Además, vamos a comparar por pares los osciladores adyacentes. Esto lo hacemos porque, como ya sabemos, la variación sistemática depende mucho de la distribución espacial y las condiciones de funcionamiento (temperatura, etc.), de forma que al comparar dos osciladores de anillo dispuestos simétricos y adyacentes, la variación en el retraso debido a la variación sistemática en ambos será muy parecida. Por lo que, al compararlos, prácticamente eliminaremos su contribución, siendo esta mínima y en la mayoría de los casos tendiendo a cero. Cabe señalar que este método tiene una desventaja clara y es que el número de bits independientes que se pueden obtener de la PUF con un número n de osciladores es (n-1).

6 PUF de osciladores de anillo (RO-PUF)

Una vez entendido el funcionamiento de un oscilador de anillo, haber realizado un diseño del mismo en Vivado, haberlo implementado en la FPGA y haber comprendido una serie de problemas que conlleva la implementación del mismo, se va a describir la arquitectura de una posible PUF basada en osciladores de anillo (RO-PUF). El flujo de trabajo va ser el mismo que seguimos para la implementación del oscilador de anillo, lo que ahora con un diseño más amplio que va a utilizar como base, el trabajo ya realizado con el oscilador de anillo. Como ya hemos adelantado en la sección anterior, la PUF se basa en la comparación de frecuencias entre pares de osciladores. Obviamente, si conseguimos eliminar la variación sistemática, la

diferencia en frecuencias se producirá únicamente por las variaciones aleatorias introducidas en el proceso de producción de la FPGA, por lo que el diseño cumplirá la condición necesaria de exclusividad.

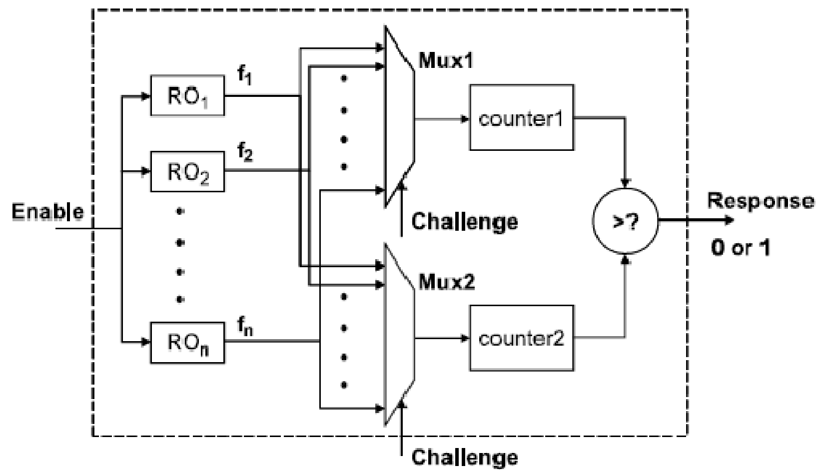

Figura 3. Arquitectura conceptual de la RO-PUF. [MAI 10]

La arquitectura conceptual de la RO-PUF propuesta es la representada en la figura 3. En primer lugar, se dispone de una señal *enable* global para activar la PUF. Vemos que la parte más importante consiste en la realización de una serie de *n* osciladores de anillo, que ubicaremos en diferentes lugares de placa. Con el fin de evitar los efectos de la variación sistemática, vamos a implementar osciladores simétricos, es decir, con la misma distribución espacial de inversores y a disponerlos en posiciones adyacentes en la matriz, de manera que conformen una especie de submatriz de osciladores (figura 4).

Figura 4. Submatriz de osciladores con configuración vertical que componen la RO-PUF.

En la figura 4, se pueden visualizar 6 osciladores distintos configurados verticalmente. Con sus respectivas puertas AND (LUT2s iluminados de abajo) y con 16 inversores (LUT1s) por oscilador de anillo que van completando los CLBs.

Todos estos osciladores de anillo llevan su salida a dos multiplexores, los cuales a su vez seleccionan dos de las salidas de todos los osciladores de anillos. Estas dos salidas seleccionadas por la señal reto corresponden a osciladores adyacentes, las cuales van a parar cada una a dos contadores respectivamente, encargados de medir sus respectivas frecuencias. Finalmente, el valor asociado a cada frecuencia es introducido en la entrada de un comparador; el cual genera un bit con valor "1" o "0" en función de si una frecuencia es más alta que otra. A continuación, se describe sucintamente la programación en Verilog de la RO PUF.

6.1 Módulos de osciladores (Anexo C.1)

Las características más relevantes de estos módulos ya han sido explicadas en la sección 5 (Implementación del oscilador de anillo). Básicamente, cada módulo está compuesto por el número de inversores que nosotros fijemos y una puerta AND, estando todos ellos unidos por variables tipo *wire*[2]. El módulo tiene una entrada tipo *enable*, la cual va a la puerta AND y permite habilitar el oscilador (tabla 1). Además, tiene una salida tipo *output-ro*, a la cual mediante una asignación continua condicional se le asigna un valor cero o se le une a la salida del último inversor, en función del valor del *enable*.

6.2 Módulo selección (Anexo C.2)

Se trata de un módulo jerárquico superior, que invoca a todos los módulos de osciladores. El objetivo del módulo es instanciar todos los módulos de osciladores, asignarles a través de su entrada *enable* el mismo valor de entrada a todos los módulos de osciladores y generar una salida del módulo tipo *wire* [N:0][3] *out-ro,* que contenga todas las salidas de los módulos oscilador.

6.3 Módulo multiplexor (Anexo C.3)

Este módulo tiene como objetivo emular el comportamiento de un multiplexor. Tiene una entrada tipo *wire* [N:0] *osc*, por la que entran todas las entradas de los osciladores, procedentes del módulo selección. También, tiene tres entradas: *change* encargada de producir el cambio en la variable *i*; la cual selecciona las salidas de dos osciladores, *resetglobal* permite el reinicio de la variable *i* y *clock* correspondiente al reloj de la FPGA. A cada cambio de la variable *change* y siempre y cuando la variable *resetglobal* valga 0 o la variable *i* no sea mayor que el número de osciladores (reinicio de la variable *i*), se le asigna proceduralmente *blocking* a la variable *i* un valor *i*+1:

```
always @(posedge change or posedge resetglobal) begin
 if (resetglobal==1 || (i>=N)) i <= 0;
 else i <= i+1;
end
```

Se puede ver que se trata de una asignación asíncrona. En cambio, la comprobación de que el valor de *i* no supere el número de osciladores se realiza síncronamente con el reloj. Por último,

[2]Todas las entradas a los módulos son de tipo *wire*, las salidas no.

[3] Se trata una variable que es un *array de wires*.

en el caso de que se hayan seleccionado ya todos los pares adyacentes de osciladores, se le asigna un valor positivo a la variable de salida *final*. Además, tiene dos salidas tipo *wire out1* y *out2* que se asignan continuamente a dos entradas de la variable *osc* como sigue:

```
assign out1 = osc [i];
assign out2 = osc [i+1];
```

Seleccionando como salidas del módulo, la salida de dos pares adyacentes de osciladores.

6.4 Módulo comparador (Anexo C.4)

Este módulo recibe como entradas las dos salidas de los osciladores seleccionados por el módulo multiplexor, así como la señal de reloj y una entrada llamada *rst* que actúa como *reset*, permitiendo reiniciar los contadores a cero. Estas dos entradas de osciladores actúan como señal de cambio para que los dos contadores definidos en este módulo aumenten su valor en una unidad por cada cambio de flanco ascendente en cada una de las dos señales. Una vez uno de los dos contadores alcanza un valor predefinido; que pasaremos a llamar como *número de cuentas,* y que se define en el módulo como un parámetro *N_ciclos*, se detiene el conteo. La comprobación de que dicho contador iguala el *número de cuentas* se realiza de forma síncrona con el reloj:

```
 always @(posedge clock) begin
if(count2 >= N_ciclos) stop = 1;
else stop = 0;
end
```

Y produce un cambio en la variable *stop*, la cual es la que detiene los contadores. Además, es la que permite entrar a comparar el resultado de los contadores. Asignamos proceduralmente *blocking* un valor positivo a las variables de salida *clave* e *igual* en función de si la comparación ha resultado en un valor mayor del contador 1 o de si ha resultado en que los valores son iguales, respectivamente.

6.5 Módulo wrapper y final (Anexo C.5)

El módulo final se encarga de que cada vez que se produzca un cambio en la selección de contadores del módulo selección, la variable *rst* del módulo comparador valga “1” y así se reinicien a cero los contadores. En cuanto al módulo *wrapper*, su función es la de un módulo jerárquico superior que invoca al resto de módulos. Tiene cuatro entradas: la señal de reloj, el *enable* global que se utiliza para detener la oscilación de los osciladores a través de la variable *enable* del módulo *selección*, un reset global que reinicie los contadores y la selección de multiplexores, y una variable de control que hasta que no tenga un valor “1” no permita el cambio a los multiplexores, ni el reinicio del proceso de conteo. Esto significa que nos permitirá mantener los bits a la salida todo el tiempo que queramos. Además, dispone de cuatro salidas: *enable_led* enciende un LED cuando se han seleccionado todos los pares de osciladores, *clave_final* y *clave_igual* nos saca el bit resultante la comparación del módulo comparador y *stop* que le permitirá a la placa Arduino saber cuándo tiene que medir los bits de salida.

Para monitorizar todo el funcionamiento de la PUF en la FPGA, utilizamos una placa Arduino NANO (Anexo D), la cual era la que recibía los bits de salida y enviaba a la FPGA las señales de *control* para cambiar a otro par de osciladores y obtener nuevos bits, y la señal de *reset global* para reiniciar todo el proceso. Para diferenciar la salida *clave_igual* a la hora de sacar los bits por pantalla, a estos siempre que sea "1", se les asigna en Arduino un "2" para saber que las frecuencias son iguales.

Por otra parte, tal y como se menciona en la Sección 4 (Flujo de diseño en Vivado), en los procesos de desarrollo que se llevan a cabo para la implementación en la FPGA a través de la herramienta Vivado, tras programar la PUF, Vivado genera lo que se conoce como una descripción *Register-transfer logic* (RTL), que no es más que un esquemático de la PUF con puertas lógicas ideales. En la figura 5, se puede ver dicha descripción.

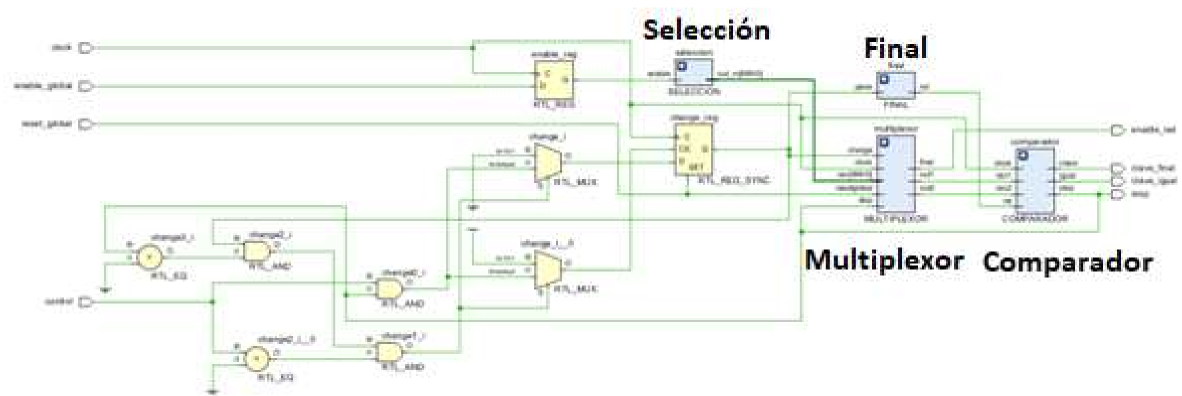

Figura 5. Descripción esquemática RTL de la RO-PUF implementada. Compuesta por 12 osciladores de anillo.

7 Parámetros característicos

Los principales parámetros [BÖM 13] empleados en la métrica de las PUF son los siguientes:

Distancia Hamming (HD) es el número de bits distintos que la nueva respuesta tiene con respecto a la que se guardó en la base de datos o la respuesta inicial (ambas utilizan el mismo reto). También se puede expresar como porcentaje, dividiendo dicho resultado para el número total de bits que tiene la cadena y multiplicando por cien. Al tratarse los outputs de cadenas de "1" o "0", solo tenemos dos opciones, que el bit sea correcto o que sea incorrecto. De manera que, estadísticamente, la distancia Hamming se comporta como una **distribución binomial** (Anexo A).

Valor medio (μ) da cuenta de la probabilidad de obtención de un "1" o un "0" para cada bit de la cadena. Este valor ya lo comentamos en la Sección 5 (Implementación del oscilador de anillo), y se puede ver alterado por la variación sistemática. **El intervalo de confianza (CI)** que tiene esta medida extendido al 95% de confianza se puede calcular como sigue:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \quad (4) \qquad\qquad CI = p \pm z\sqrt{\frac{p(1-p)}{n}} \quad (5)$$

Donde n es el número de bits de la respuesta, p es la probabilidad de ocurrencia y $z = 1 - \alpha/2$, donde en este caso $\alpha = 5\%$.

Distancia Hamming intra-chip (HD_{intra}) es la distancia Hamming entre la ID guardada y las nuevas respuestas generadas por la misma PUF con el mismo reto.

Distancia Hamming inter-chip (HD_{inter}) es la distancia Hamming entre la ID guardada de una PUF A y las nuevas respuestas generadas por distintas PUF con el mismo reto.

***False Acceptance Rate* (FAR)** es el tanto por ciento de falsos aceptados, es decir, el tanto por ciento de IDs que han sido aceptadas siendo que la PUF que la proporcionaba no era la PUF que genero la ID inicial. La fórmula que la describe es:

$$FAR = \frac{NFA}{NAA} \cdot 100\ (\%) \quad (6)$$

Donde NFA es el número de falsos aceptados y NAA es el número de intentos de ataque, entendido por ataque como el número de veces que se ha intentado colar la ID falsa.

***False Rejection Rate* (FRR)** es el tanto por ciento de casos correctos que hemos rechazado, es decir, el tanto por ciento de IDs que han sido rechazas siendo que la PUF que las proporcionaba era la PUF que genero la ID inicial. Su fórmula es:

$$FRR = \frac{NFR}{NIA} \cdot 100\ (\%) \quad (7)$$

Donde NFR es el número de IDs correctas rechazadas y NIA es el número de intentos de identificación.

Eficiencia energética (E/bit) es la energía consumida por bit generado en la PUF.

El objetivo siempre que se desarrolla una PUF, es que se ajuste lo máximo posible a lo que sería la idealidad de la misma. En la tabla 2, se pueden ver los valores que deberían tomar esos parámetros si la PUF fuera ideal:

PUF ideal

$\boldsymbol{\mu}$	$0{,}5 \pm CI$
$\boldsymbol{HD_{intra}}$	$0\ \%$
$\boldsymbol{HD_{inter}}$	$50\ \%, \sigma = 100\% \frac{\sqrt{N}}{2}$
$\boldsymbol{E/bit}$	$0\ pJ/bit$

PUF real [BÖM 13]

$\boldsymbol{\mu}$	$0{,}504 \pm 0{,}0153$
$\boldsymbol{HD_{intra}}$	$4{,}6392\ \%$
$\boldsymbol{HD_{inter}}$	$50{,}0271\ \%$
$\boldsymbol{E/bit}$	$391{,}5\ pJ/bit$

Tabla 2. Valores de los parámetros característicos para una PUF ideal y real.

Es fácil comprender el porqué de los valores de la PUF ideal de la tabla 2. En primer lugar, si queremos una aleatoriedad óptima en los bits es necesario una probabilidad de 0,5 de obtener uno u otro, tal y como vimos ya con anterioridad. En el caso de las distancias intra-chip e inter-chip, queremos que estén lo más alejadas posibles para evitar fallas en la identificación; FRR y

FAR (Anexo B). Por lo que idealmente queremos que no haya distancia intra-chip, obteniendo así siempre para el mismo PUF la misma respuesta para el mismo reto. Y que la distancia inter-chip sea un 50% del total (el total sería todos los bits distintos), ya que, si todos los PUF del mismo tipo generan bits aleatoriamente con una probabilidad de 0,5, es fácil ver que la variación media de la distancia Hamming entre outputs de PUF distintas debe ser del 50%. Por último, se tendría la eficiencia energética, que idealmente debería ser cero, aunque no es posible nunca llegar a tal valor. De forma que la calidad de la PUF generada, se puede basar en la cercanía relativa que los parámetros de dicha PUF presentan frente a los ideales.

Otro parámetro a tener en cuenta en diversas aplicaciones, es la temperatura. Para nuestro caso particular de RO-PUF, la temperatura puede llegar a ser un factor determinante. Generalmente, las frecuencias de los osciladores de anillo varían bastante con la temperatura, hecho que nos va a perjudicar ya que puede afectar a la estabilidad de las respuestas. Debido a que la generación de las respuestas se produce por la comparación entre frecuencias, el cambio de temperatura puede no afectar si produce el mismo cambio en frecuencia para todos los osciladores. Generalmente, no puede garantizarse esta compensación, por lo que al variar la temperatura habrá que evitar llegar al punto límite en el que la comparación entre frecuencias de dos osciladores cambie. Este efecto se puede observar en la figura 6.

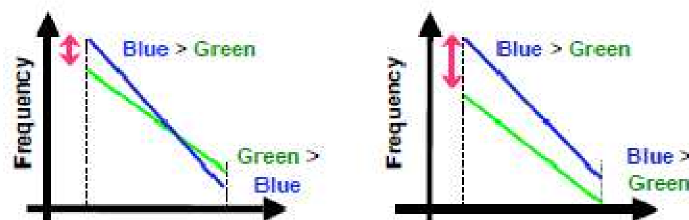

Figura 6. Efecto de la temperatura en los pares de osciladores. [SUH 07]

Es por ello que en casos en los que la temperatura sufra amplias variaciones, es bueno realizar una comparativa de osciladores con frecuencias lo más alejadas posibles para evitar que se cambie el bit al variar la temperatura.

8 Caracterización experimental

8.1 Caracterización del oscilador de anillo

Para la comprobación de la variación sistemática comentada en la Sección 5 (Implementación del oscilador de anillo), se implementó una misma topología de oscilador de anillo con todos sus inversores dispuestos verticalmente y otra con todos sus inversores dispuestos de forma que se vayan llenando las CLBs horizontalmente (figura 7).

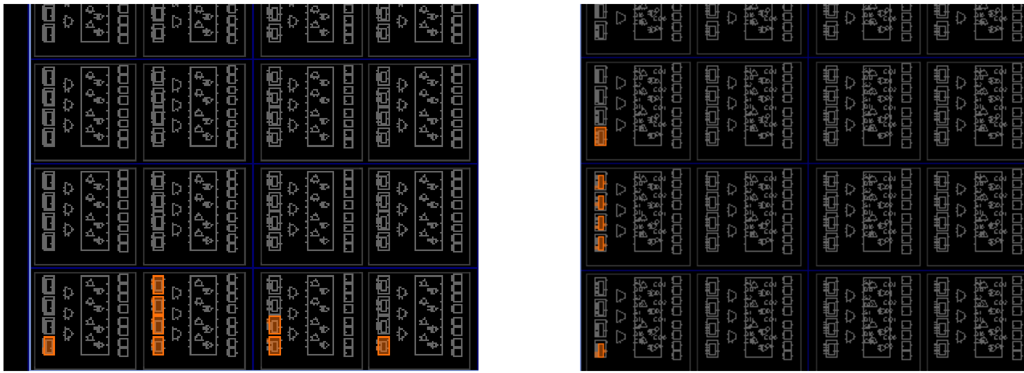

Figura 7. Configuración horizontal (izquierda) y vertical (derecha) de un oscilador.

Además, con el fin de poder comprobar la variación en frecuencia del oscilador de anillo en función del número de inversores (debido a los retrasos de los mismos) se midieron para distinto número de inversores la frecuencia del oscilador de anillo, tanto con la disposición vertical de inversores como con la horizontal, expuestas anteriormente. Los resultados se presentan en la figura 8.

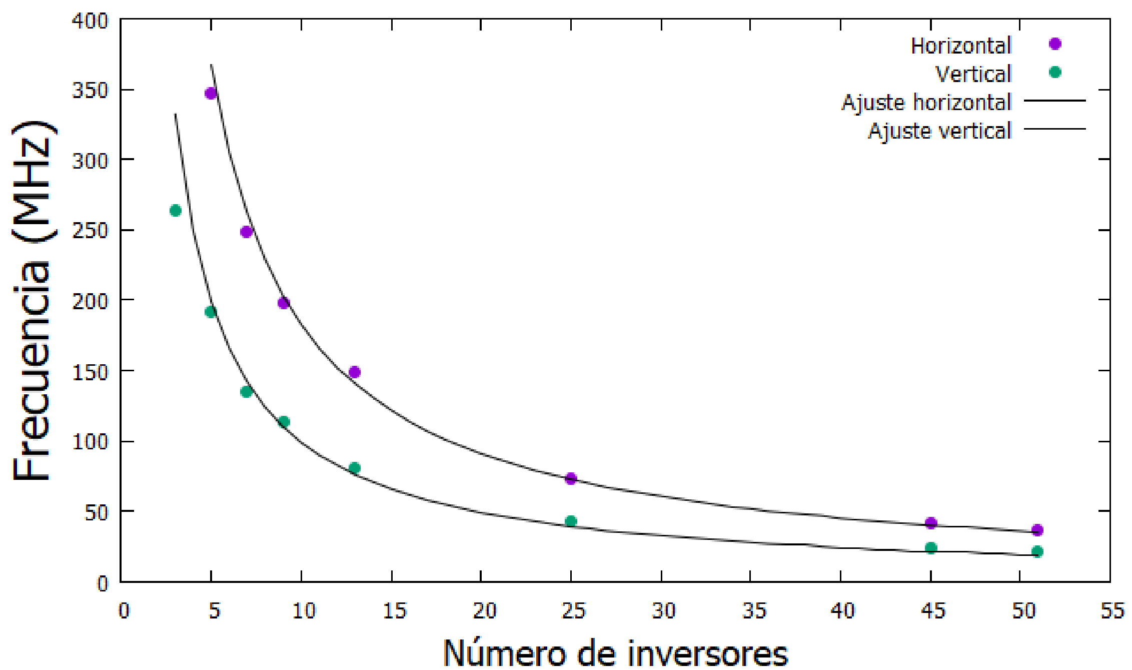

Figura 8. Gráfica de las medidas experimentales de frecuencia en función del número de inversores obtenidas para la configuración horizontal y vertical.

Es fácil visualizar en la figura 8 los dos efectos comentados anteriormente, en primer lugar, vemos que la variación sistemática es apreciable entre ambas disposiciones. Los osciladores con inversores dispuestos verticalmente alcanzan frecuencias sustancialmente menores, siendo más notable esta variación conforme el número de inversores va disminuyendo. Además, se puede observar también como se ajustan a la fórmula (1) ya que al aumentar el número de inversores decae la frecuencia. Obtenemos el siguiente valor promedio del retardo por inversor:

Configuración vertical	Configuración horizontal
$t_p = 0{,}5015\ \text{ns}$	$t_p = 0{,}2725\ \text{ns}$

8.2 Caracterización de la PUF de osciladores de anillo (RO-PUF)

Con el objetivo de comprobar el funcionamiento y la calidad de la RO-PUF desarrollada, se realizaron una serie de medidas. En primer lugar, tratamos de ver como la variación en el número de inversores de cada oscilador afectaba al comportamiento de la RO-PUF, ya que esta variación supone un cambio en frecuencia de cada oscilador (figura 8). Por tanto, fijamos un número de 12 osciladores para el desarrollo de estas medidas, y tomamos datos para las dos configuraciones de anillo (figura 7), así como para valores de 1000 y 10000 *número de cuentas* realizadas por el contador en función de la frecuencia.

Del análisis de estos datos, se puede ver en el caso de la configuración horizontal cómo al aumentar el número de inversores por oscilador, la tendencia es un aumento de la distancia intra-chip, así como la aparición de un mayor número de estados indeterminados X (frecuencias iguales). Esto se debe a que al aumentar el número de inversores, cada uno de ellos con su consiguiente variación aleatoria, tienden a compensarse dichas variaciones, reduciendo por tanto su contribución. De forma que los osciladores tienen frecuencias más cercanas, provocando una generación de IDs mucho menos estable para *número de cuentas* bajo, es decir, sería necesario aumentar mucho el *número de cuentas*. En la tabla 3, se puede ver el aumento del número de indeterminaciones con el número de inversores:

N° inversores por oscilador	ID Horizontal	ID Vertical
5	11010101110	01010101010
9	11101101110	01010101010
21	11001101100	01010101010
81	110X01001X0	01010101010
141	111X01001X1	01010101010

Tabla 3. Respuestas de la RO-PUF para la configuración horizontal y vertical en función del número de inversores por oscilador.

Cabe señalar que en estas medidas y en las siguientes, tomamos muchas IDs para una misma configuración de la PUF y tomamos como ID de la misma, aquella que fuera la moda matemática de todas ellas.

Por otro lado, el análisis de la configuración vertical nos arroja un resultado distinto. Se obtiene que para cualquier número de inversores por oscilador el valor de la distancia Hamming intra-chip es cero, es decir, obtenemos siempre la misma ID. A pesar de lo satisfactorio que podría ser dicho resultado, si se profundiza un poco más en él, se observa que esta falta de cambio en la ID se debe a la variación sistemática. De forma que la configuración

vertical sufre de una variación sistemática importante, debido a que este modelo de FPGA ofrece una estructura menos simétrica al oscilador si se dispone en dicha configuración. Esta variación sistemática es un problema para el desarrollo de una PUF, como ya hemos comentado con anterioridad. Es por ello que esta configuración se descarta para la implementación de la PUF.

Además, debido a la reducción de frecuencia por el aumento del número de inversores, el tiempo de generación de la ID se ve aumentado. Ya que les cuesta más a los contadores llegar al *número de cuentas* que detiene el conteo y proceder a la comparación de los mismos. Es por ello, que seleccionaremos para el resto de medidas un número de inversores por oscilador de 5, ya que tiene una frecuencia muy elevada que permite una rápida generación de la ID y además permite que la variación aleatoria final sea notable y no se vea atenuada por compensación estadística.

Una vez establecido el número de inversores, vamos a tratar de determinar si *el número de cuentas* que realizan los contadores en cada toma de medidas se puede reducir, con el fin de aumentar la rapidez en la generación de la respuesta. Para ello, se fijó la RO-PUF a 5 inversores por oscilador y un número total de 12 osciladores. El objetivo es ver cómo se comporta la distancia HD_{intra} con el *número de cuentas*, para así poder seleccionar el mínimo *número de cuentas* que nos dé una distancia intra-chip aceptable y consumir así el menor tiempo posible en la obtención de cada bit. De forma que se realizaron medidas de 10000 IDs con los valores de 10, 100, 1000 y 10000 *número de cuentas* para una disposición horizontal de los osciladores.

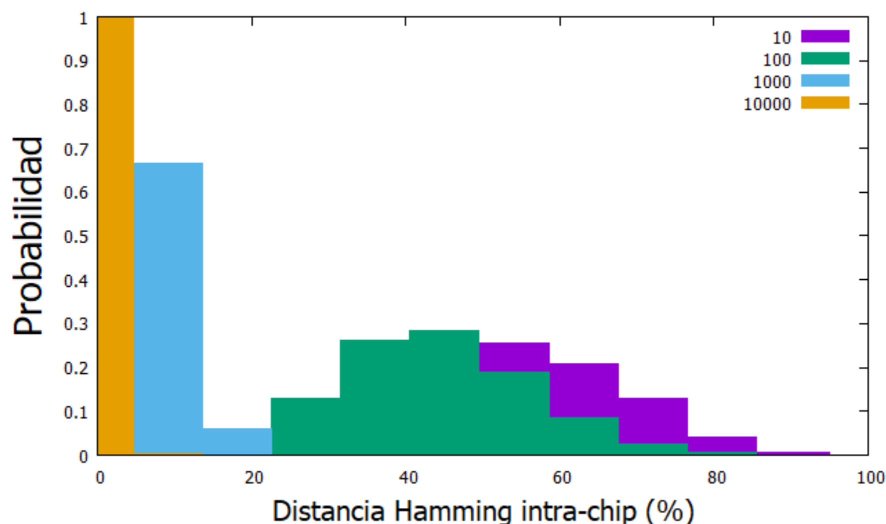

Figura 9. Distancia Hamming intra-chip para diferentes valores del *número de cuentas*.

En la figura 9, se ve claramente como la distancia intra-chip se reduce mucho para los valores de 1000 y 10000, lo que es un buen indicador, ya que nos permitirá realizar una buena identificación para dichos valores. En cambio, para valores bajos del *número de cuentas* se produce un aumento de la distancia intra-chip, esto se debe a que el pequeño periodo de recuento no permite que las pequeñas variaciones aleatorias de frecuencia tengan un efecto

notable a la salida de los contadores. Además, se puede ver como para un valor de 10, obtenemos muchos bits indeterminados tras el procesamiento del Arduino, lo que nos indica que ambos contadores han ofrecido el mismo resultado debido al bajo tiempo de conteo, en concreto la ID obtenida es X1XXXXX11XX. Para el caso de 100, también se obtienen, aunque en menor medida, un gran número de indeterminaciones XX00X0X0110.

Obtenemos por tanto un primer resultado, que consiste en el descarte del valor 10 y el 100 como final del proceso de conteo. En cuanto a la selección de uno de los otros dos valores, como veremos a continuación depende de la posición de la PUF en la placa. A continuación, para poner en valor la característica más destaca de la PUF, que consiste en la generación de distintas IDs por distintos dispositivos, como no disponemos de un número suficiente de FPGAs, disponemos los osciladores idénticos en distintos lugares de la FPGA (figura 10).

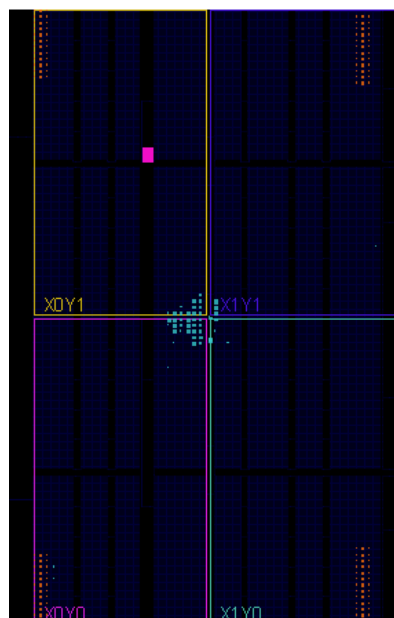

Figura 10. Posición de las distintas PUF implementadas en la matriz de la FPGA.

De forma que, a efectos prácticos, cada una de las cuatro disposiciones de las esquinas de la FPGA actúa como una PUF distinta al resto, emulando la situación de su ubicación en cuatro dispositivos diferentes. De nuevo, debido a que todas las PUF implementadas con configuraciones verticales daban una alta variación sistemática, tomamos en cuenta solo la configuración horizontal de inversores, tal y como se puede observar en la figura 10. Tomando como referencia las conclusiones extraídas de las medidas realizadas anteriormente, fijamos a 5 inversores los osciladores, formamos la PUF con 12 osciladores y tomamos medidas para 1000 y 10000 *número de cuentas*. En la tabla 4, podemos ver las IDs de las distintas posiciones:

Posición en la FPGA	ID	HD_{inter}
X0Y0 $\rightarrow x\,\epsilon[1,3]\;\; y\,\epsilon[1,12]$	11010101110	
X0Y1 $\rightarrow x\,\epsilon[1,3]\;\; y\,\epsilon[87,98]$	10011011010	45,45 %
X1Y0 $\rightarrow x\,\epsilon[40,42]\;\; y\,\epsilon[1,12]$	11001101010	27,27 %
X1Y1 $\rightarrow x\,\epsilon[40,42]\;\; y\,\epsilon[87,98]$	00110101100	36,36 %

Tabla 4. Respuestas de la RO-PUF para 10000 cuentas y para las distintas posiciones M_{xy} de la matriz donde se ha implementado.

Se observa claramente la variación inter-chip en las IDs, lo que es un indicador óptimo del buen funcionamiento de la PUF desarrollada. En el caso del *número de cuentas* seleccionado, se utiliza en general 1000, aunque en función de la posición en la matriz, a veces es necesario aumentarlo a 10000 para disminuir la distancia Hamming máxima intra-chip (tabla 5).

Posición en la FPGA	HD_{intra} máxima 1000	HD_{intra} máxima 10000
X0Y0 $\rightarrow x\,\epsilon[1,3]\;\; y\,\epsilon[1,12]$	0%	0%
X0Y1 $\rightarrow x\,\epsilon[1,3]\;\; y\,\epsilon[87,98]$	18,18%	0%
X1Y0 $\rightarrow x\,\epsilon[40,42]\;\; y\,\epsilon[1,12]$	0%	0%
X1Y1 $\rightarrow x\,\epsilon[40,42]\;\; y\,\epsilon[87,98]$	27,27 %	18,18%

Tabla 5. Máxima distancia Hamming intra-chip de la RO-PUF para las distintas posiciones M_{xy} de la matriz donde se ha implementado y *número de cuentas*.

Vemos por tanto que de cada FPGA se puede sacar un elevado número de IDs de diferentes tamaños y valores, lo cual será de especial utilidad para la autenticación de la FPGA mediante el sistema de *Challenge-response pairs* (CRP). Cabe resaltar que en el proceso de medida se pudo observar que era preferible no ubicar osciladores en las CLBs posicionadas en el perímetro de la matriz para conseguir una mayor estabilidad en los casos.

Por último, realizamos la comparación entre dos FPGA distintas del mismo modelo. El objetivo es comprobar cómo implementando exactamente el mismo diseño y ubicación de PUF en ambas, la ID obtenida es diferente. Para ello, aumentamos el número de osciladores a 33 para una visualización más clara del efecto, y tomando en cuenta todos los resultados obtenidos anteriormente generamos la RO-PUF en ambas matrices. Tomamos 10000 como el *número de cuentas*, para realizar una medida más precisa, en la tabla 6 se pueden ver las IDs obtenidas:

FPGA 1	FPGA 2
0010011010100101111001001011011 1	11001101010011000101010110100111
	$HD_{inter} = 48{,}48\ (\%)$

Tabla 6. Respuestas de la RO-PUF implementada en dos FPGAs diferentes.

Una vez fijados los valores más correctos para la RO-PUF y comprobada la generación de distintas IDs, vamos a valorar la exclusividad y reproducibilidad de la misma. Para ello, compararemos la PUF propuesta con los valores de la PUF ideal. En primer lugar, vamos a calcular las distancias Hamming intra-chip (reproducibilidad) y Hamming inter-chip (exclusividad). Para ello, vamos a utilizar la premisa de que si implementamos la RO-PUF en distintas partes de la FPGA, estas se pueden considerar PUFs distintas, debido a que no disponemos de suficientes FPGAs para realizar una buena estadística de la distancia inter-chip. Por tanto, fijamos el número de inversores de cada oscilador a 5, una configuración de los mismos horizontal y un *número de cuentas de* 1000. Generamos nuestras RO-PUF con estos datos, las cuales ocupan cada una, una fila de matriz de la FPGA y están compuestas por 14 osciladores. Los resultados se presentan en la figura 11.

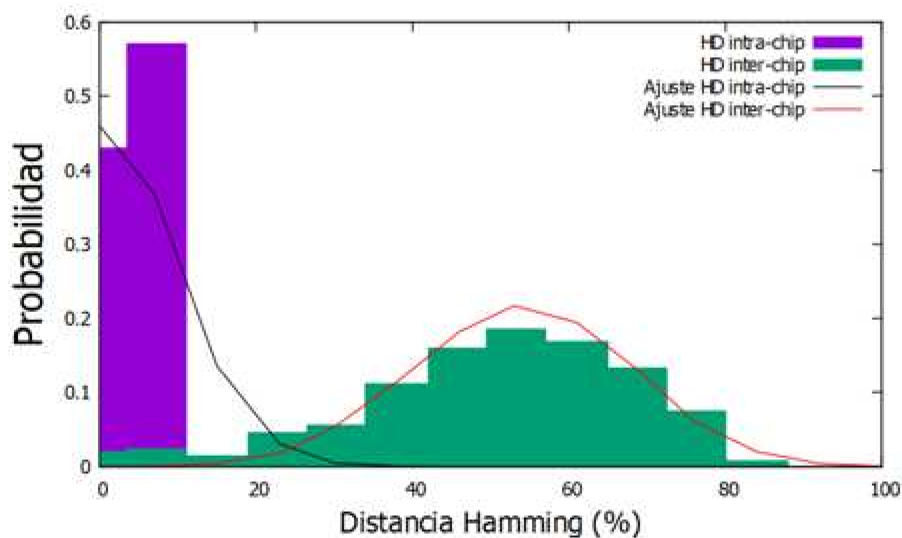

Figura 11. Distancias Hamming intra-chip e inter-chip de la RO-PUF implementada.

Ajustando dichos resultados a una binomial (figura 11), podemos obtener los valores de las distancias Hamming intra-chip e inter-chip:

$$HD_{intra} = 4{,}39\ \% \qquad HD_{inter} = 50{,}84\ \%$$

$$\sigma_{intra} = 3{,}81\ \% \qquad \sigma_{inter} = 17{,}77\ \%$$

Y a partir de ellos calcular, mediante la FAR y FRR (Anexo B), la distancia Hamming máxima de identificación. De forma que si la distancia Hamming de la nueva respuesta frente a la ID guardada es menor que dicha distancia Hamming máxima se da por identificado.

Normalmente esta distancia Hamming máxima se dispone de manera que la FAR y la FRR sean lo más iguales posibles:

$$HD_{max} = 3$$

Obteniendo unos valores de:

$$FRR = 3{,}60\ \% \qquad\qquad FAR = 2{,}26\ \%$$

Vivado nos ofrece una estimación de la potencia consumida. Obviamente dentro de esta potencia entran muchos factores como el número de registros y conectores utilizados, el reloj de la FPGA, las salidas y entradas, etc. El valor estimado de potencia para la configuración de RO-PUF utilizada es de 0,005 W (incluidos recursos compartidos), si lo multiplicamos por el tiempo estimado que le cuesta obtener a esta RO-PUF cada bit 2,88 $\mu s/bit$, obtenemos que la eficiencia energética es 14,4 nJ/bit. A pesar de ser algo alta en comparación con el valor de la PUF real de la tabla 2 (que no del tipo RO-PUF), cabe señalar que no se ha realizado un proceso de optimización de la eficiencia energética, por lo que se podría mejorar dicho valor.

Por último, se puede ver en la figura 12, el puesto de trabajo y su diagrama de bloques.

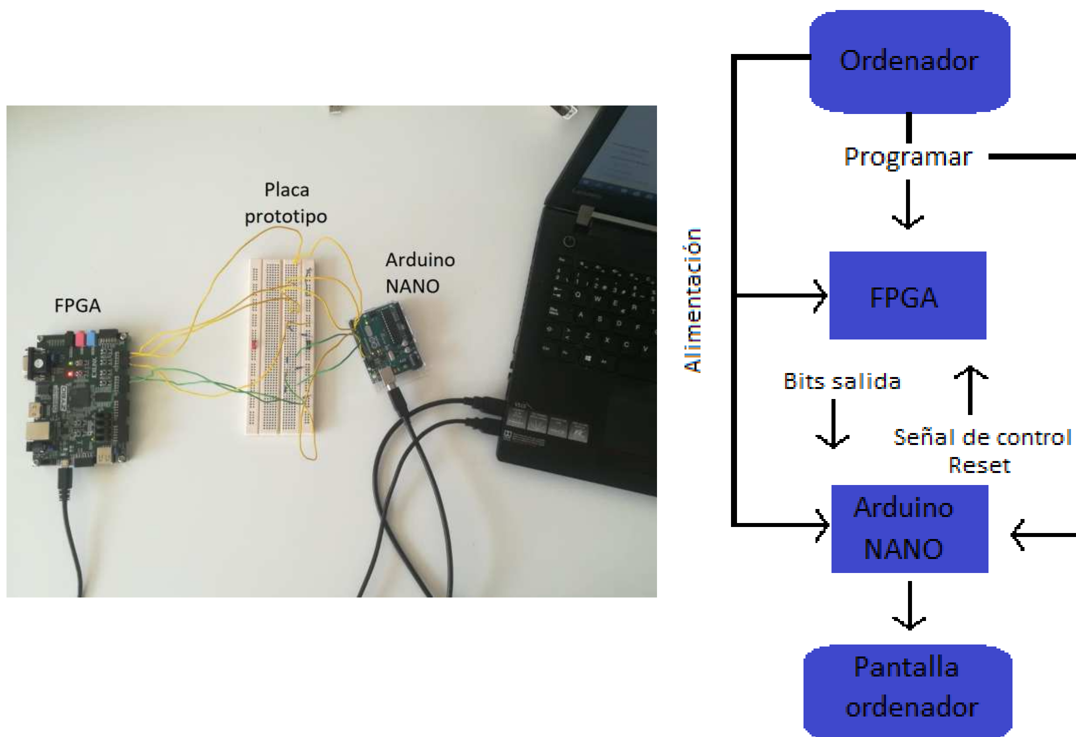

Figura 12. Puesto de trabajo y diagrama de bloques.

9 Conclusión

El TFG se ha centrado en la introducción a las funciones físicas no clonables (PUF) mediante estructuras implementables en circuitos integrados, a través de una primera inmersión en el ámbito teórico y aplicaciones: la identificación, autenticación y generación de claves. Además, se ha diseñado y desarrollado un tipo concreto de estos dispositivos: la PUF basada en

osciladores de anillo; y se ha llevado a cabo una caracterización experimental preliminar de las estructuras integradas en FPGA.

Las conclusiones extraídas sobre la PUF generada son positivas, aunque como se ha ido desarrollando a lo largo del trabajo, esta tiene una serie de limitaciones, entre la que destaca su debilidad a posibles ataques debido al bajo número de *Challenge-response pairs* (CRP). Limitando sus aplicaciones al ámbito principalmente de identificación, sin la necesidad de post procesado de la respuesta y posiblemente también a aplicaciones de autenticación, pero en el concurso de códigos de corrección de errores.

Quiere hacerse constar, que este TFG ha requerido del estudio de los fundamentos de la electrónica digital, de un adiestramiento en programación de FPGAs a través de la aplicación Vivado y el aprendizaje del lenguaje de descripción hardware Verilog, suponiendo una dedicación de tiempo y esfuerzo considerables.

10 Bibliografía

[BÖM 13] Böhm, C., Hofer, M., *Physical Unclonable Functions in Theory and Practice,* Springer-Verlag, New York, (2013)

[DEV 11] Devadas, S., *Practical Applications of Physical Unclonable Functions (PUFs),* (2011)

[HAN 11] Handschuh, H., Schrijen, G. J., Tuyls, P., *Hardware Intrinsic Security from Physically Unclonable Functions,* Springer-Verlag, (2011)

[HER 14] Herder, C., Yu, M. D., Koushanfar, F., Devadas, S., *Physical Unclonable Functions and Applications: A Tutorial,* IEEE, (2014)

[MAE 12] Maes, R., *Physical Unclonable Functions: Constructions, Properties and Applications,* (2012)

[MAI 10] Maiti, A., Schaumont, P., *Improved Ring Oscillator PUF: An FPGA-friendly Secure Primitive*, (2010)

[MAX 04] Maxfield, C., *Design Warrior´s Guide to FPGAs*, Mentor Graphics Corporation and Xilinx, (2004)

[NVT 13] *Nexys4 Vivado Tutorial-25*, Xilinx, (2013)

[RAB 03] Rabaey, J. M., *Digital Integrated Circuits: A Design Perspective,* Prentice Hall, (2003)

[RUH 14] Ruhrmair, U., Holcomb, D. E., *PUFs at a Glance*, EDAA (2014)

[STFC 16] Microelectronics Support Centre STFC Rutherford Appleton Laboratory, *Introduction to Verilog and System Verilog for Design*, (2016)

[SUH 07] Suh, G. E., Devadas, S., *Physical Unclonable Functions for Device Authentication and Secret Key Generation,* (2007)

Universidad Zaragoza

Trabajo Fin de Grado:

Anexos

Diseño CMOS de funciones físicas no clonables

Autor

Rubén Martín Pinardel

Directores

Santiago Celma Pueyo

Miguel García Bosque

Departamento de Ingeniería Electrónica y Comunicaciones

Junio 2019

Anexo A. Distribución de Bernoulli y binomial

La distribución de Bernoulli se trata de una distribución de probabilidad discreta, en la que la variable a medir puede tomar únicamente dos valores. Estos dos valores se pueden tomar clásicamente como la probabilidad de éxito o fracaso, y en nuestro caso más particular relacionado con la electrónica puede tomar el valor "1" o "0". Cada uno de los dos valores tiene una probabilidad, las cuales están obviamente relacionadas ya que son complementarias. De forma que si el valor asignado al éxito o "1" tiene una probabilidad normalizada a uno de p, el otro valor tendrá una probabilidad que es $q = (1 - p)$. Es obvio por tanto que la función de probabilidad de la variable se puede expresar como:

$$f(x) = p^x (1-p)^{1-x}$$

Donde *x* es la variable. Si ahora tenemos muchas variables iguales que esta variable *x*, completamente independientes entre sí, tal y como nos sucede en nuestro TFG con los *arrays* de bits, se distribuyen en lo que se conoce como distribución binomial.

La distribución binomial se trata de un conjunto de distribuciones de Bernoulli independientes. De forma que la variable *X* que se distribuye como una binomial tiene en cuenta el número de éxitos o "1" que se obtienen tras *n* experimentos de Bernoulli. Cabe resaltar que aunque son independientes, todos comparten una probabilidad fija de éxito p. De forma que la función de probabilidad de la variable *X* tras *n* experimentos de Bernoulli es:

$$f(X) = \binom{n}{X} p^X (1-p)^{n-X}$$

Donde se tienen en cuenta el número de combinaciones de *x* elementos extraídos de n. Claramente para nuestro caso, los experimentos de Bernoulli se tratarían de bits obtenidos tras aplicar un reto a nuestra PUF. Además es fácil demostrar que la media de esta función tiene el siguiente valor:

$$E(X) = np$$

Anexo B. Cálculo distancia Hamming máxima

En primer lugar, la conocida como "False Acceptance Rate" (FAR), se trata del tanto por ciento de falsos aceptados, es decir, del tanto por ciento de IDs que han sido aceptadas siendo que la PUF que la proporcionaba no era la PUF que generó en la ID inicial. La fórmula que la describe es:

$$FAR = \frac{NFA}{NAA} \cdot 100\ (\%)$$

Donde NFA es el número de falsos aceptados y NAA es el número de intentos de ataque, entendido por ataque como el número de veces que se ha intentado colar la ID falsa.

Por otro lado la conocida como "False Rejection Rate" (FRR), se trata del tanto por ciento de casos correctos que hemos rechazado, es decir, del tanto por ciento de IDs que han sido rechazas siendo que la PUF que las proporcionaba era la PUF que genero la ID inicial. Su fórmula es:

$$FRR = \frac{NFR}{NIA} \cdot 100\ (\%)$$

Donde NFR es el número de IDs correctas rechazadas y NIA es el número de intentos de identificación.

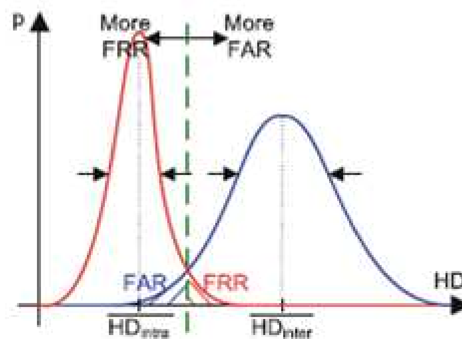

Figura 13. Distribución binomial de las distancias Hamming intra-chip (roja) e inter-chip (azul), así como representación de la distancia Hamming máxima.

En la figura 13, podemos ver las distribuciones binomiales generadas tras realizar medidas de las distintas respuestas, procedentes de lo que sería el mismo PUF (HD_{intra}) que género la ID y otro PUF (HD_{inter}), así como la HD_{max} para la que aceptaremos IDs. En este caso, tanto la FAR como la FRR pueden ser obtenidas como:

$$FRR = \sum_{i=0}^{HD_{max}} HD_{inter}(i)\ \ (\%)$$

$$FAR = \sum_{i=HD_{max}}^{HD_{Total}} HD_{intra}(i)\ \ (\%)$$

Donde $HD_{inter}(i)$ es la binomial generada por las medidas HD_{inter}, $HD_{intra}(i)$ es la binomial generada por las medidas HD_{intra}, *i* da cuenta de la variable X de la binomial dependiente de la distancia Hamming y HD_{Total} es la distancia Hamming máxima para una palabra de N bits.

Es fácil ver por tanto que la FRR será la parte de la binomial de la HD_{intra} que sobrepase la HD_{max}, provocando que sea falsamente rechazada. Y la FAR será la parte de la binomial de la HD_{inter} que está a menor distancia Hamming que la HD_{max}, siendo por tanto falsamente aceptada. Es por ello que para que haya un equilibrio entre ambos factores, a la hora de establecer la distancia HD_{max} se tiene en cuenta que la FRR y la FAR sean iguales.

Anexo C. Códigos Verilog

C.1 Código de dos módulos correspondientes a los osciladores de anillo de la RO-PUF

```
module RO_0(enable, output_ro);
    input enable;
    output output_ro;
    (* ALLOW_COMBINATORIAL_LOOPS = "true", KEEP = "true" *) wire out_ro;
    wire [0:4] w;

    assign output_ro = (enable)? out_ro : 1'b0;

    (* BEL="A6LUT", LOC="SLICE_X0Y0", DONT_TOUCH="true" *) LUT2 #(4'b1000) AND(.O(w[0]), .I0(enable), .I1(out_ro));
    (* BEL="A6LUT", LOC="SLICE_X1Y0", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_0(.O(w[1]), .I0(w[0]));
    (* BEL="B6LUT", LOC="SLICE_X1Y0", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_1(.O(w[2]), .I0(w[1]));
    (* BEL="C6LUT", LOC="SLICE_X1Y0", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_2(.O(w[3]), .I0(w[2]));
    (* BEL="D6LUT", LOC="SLICE_X1Y0", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_3(.O(w[4]), .I0(w[3]));
    (* BEL="A6LUT", LOC="SLICE_X2Y0", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_4(.O(out_ro), .I0(w[4]));

endmodule

module RO_1(enable, output_ro);
    input enable;
    output output_ro;
    (* ALLOW_COMBINATORIAL_LOOPS = "true", KEEP = "true" *) wire out_ro;
    wire [0:4] w;

    assign output_ro = (enable)? out_ro : 1'b0;

    (* BEL="A6LUT", LOC="SLICE_X0Y1", DONT_TOUCH="true" *) LUT2 #(4'b1000) AND(.O(w[0]), .I0(enable), .I1(out_ro));
    (* BEL="A6LUT", LOC="SLICE_X1Y1", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_0(.O(w[1]), .I0(w[0]));
    (* BEL="B6LUT", LOC="SLICE_X1Y1", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_1(.O(w[2]), .I0(w[1]));
    (* BEL="C6LUT", LOC="SLICE_X1Y1", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_2(.O(w[3]), .I0(w[2]));
    (* BEL="D6LUT", LOC="SLICE_X1Y1", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_3(.O(w[4]), .I0(w[3]));
    (* BEL="A6LUT", LOC="SLICE_X2Y1", DONT_TOUCH="true" *) LUT1 #(2'b01) inv_4(.O(out_ro), .I0(w[4]));
```

C.2 Código del módulo selección de la RO-PUF

```
module SELECCION(enable, out_ro);
    input enable;
    output out_ro;
       wire [11:0] out_ro;

    RO_0 ring_oscillator_0 (enable, out_ro [0]);
    RO_1 ring_oscillator_1 (enable, out_ro [1]);
    RO_2 ring_oscillator_2 (enable, out_ro [2]);
    RO_3 ring_oscillator_3 (enable, out_ro [3]);
    RO_4 ring_oscillator_4 (enable, out_ro [4]);
    RO_5 ring_oscillator_5 (enable, out_ro [5]);
    RO_6 ring_oscillator_6 (enable, out_ro [6]);
    RO_7 ring_oscillator_7 (enable, out_ro [7]);
    RO_8 ring_oscillator_8 (enable, out_ro [8]);
    RO_9 ring_oscillator_9 (enable, out_ro [9]);
    RO_10 ring_oscillator_10 (enable, out_ro [10]);
    RO_11 ring_oscillator_11 (enable, out_ro [11]);


endmodule
```

C.3 Código del módulo multiplexor de la RO-PUF

```
module MULTIPLEXOR (clock, osc, change, resetglobal, final, out1, out2);
    input osc, change, resetglobal, clock;
    output out1, out2, final;
       wire out1, out2;
       wire [11:0] osc;
       reg [11:0] i=0;
       reg final;
       parameter N=10;

    always @(posedge change or posedge resetglobal) begin
     if (resetglobal==1 || (i>=N)) i <= 0;
     else i <= i+1;
    end

    always @(clock) begin
     if (i>=N) final <= 1;
     else final <= 0;
    end

   assign out1 = osc [i];
   assign out2 = osc [i+1];


endmodule
```

C.4 Código del módulo comparador de la RO-PUF

```
module COMPARADOR (osc1, osc2, clock, rst, stop, clave, igual);
    input osc1, osc2, clock, rst;
  output stop, clave, igual;
   reg clave=0, igual=0, reset;
   parameter N_ciclos = 1000;
   reg stop = 0;
   integer count1 = 0;
   integer count2 = 0;

    always @(posedge osc1  or posedge rst) begin
   if(rst)  count1 <= 0;
   else if( rst == 0 && stop == 0) count1 <= count1 + 1;
    end


    always @(posedge osc2  or posedge rst) begin
   if(rst)  count2 <= 0;
   else if( rst == 0 && stop == 0) count2 <= count2 + 1;
    end


    always @(posedge stop) begin
    if(count1 > count2) begin
     clave <= 1;
     igual <= 0;
    end
    if(count1 < count2) begin
     clave <= 0;
     igual <= 0;
    end
    if(count1 == count2) begin
     igual <= 1;
     clave <= 0;
    end
   end

     always @(posedge clock) begin
    if(count2 >= N_ciclos) stop = 1;
    else stop = 0;
    end


 endmodule
```

C.5 Código del módulo wrapper y final de la RO-PUF

```
module wrapper(clock, enable_global, reset_global, control, enable_led, clave_final, clave_igual, stop);
    input clock;
    input enable_global, control, reset_global;
   output enable_led;
    output clave_final, clave_igual, stop;
    wire clave_final, enable_global, reset_global, clave_igual;
    wire  out1, out2;
    wire [11:0] out_osc;
    reg enable=0;
    reg change=0;
    wire stop, reset;


    always @(posedge clock) begin
   if(enable_global) enable <= 1;
   else enable <= 0;

   if(change == 1 && stop ==0 && control == 0) change <= 0;
   else if(control == 1 && stop ==1) change <= 1;

   if(reset_global) change <= 1;
    end


    SELECCION seleccion (enable,out_osc);
    MULTIPLEXOR multiplexor (clock, out_osc, change, stop, reset_global, enable_led, out1, out2);
    COMPARADOR comparador (out1, out2, clock, reset, stop, clave_final, clave_igual);
    FINAL final (change, reset);



endmodule
```

```
module FINAL (parar, rst);
    input parar;
   output rst;

   assign rst = (parar)? 1'b1 : 1'b0;


endmodule
```

Anexo D. Código Arduino NANO

```
#define N_claves 100
#define nBit 32

unsigned int i=0,j=0,k=0, contador=0, comp, igual, fin, result;

void setup() {
  //put your setup code here, to run once:
  pinMode(2, INPUT); //clave_final
  pinMode(3, INPUT); //clave_igual
  pinMode(4, INPUT); //stop

  pinMode(5, OUTPUT); //reset_global
  pinMode(6, OUTPUT); //control (avanza)

  Serial.begin(9600);
}

void loop() {
    // put your main code here, to run repeatedly:

  digitalWrite(5, HIGH);
  digitalWrite(5, LOW); //reseteo

  Serial.print("-"); Serial.print(" "); Serial.println(nBit); Serial.flush();
  for(i=0;i<N_claves;i++)
  {
    for(j=0;j<nBit;j++)
    {
       fin=digitalRead(4);

   if(fin==1)
   {

      comp = digitalRead(2);
      igual = digitalRead(3);
      result = 2*igual+comp;

      if(result==3)
      {
        result = 2;
      }

      Serial.print(result);
      Serial.print(" ");
      Serial.flush();


     digitalWrite(6,HIGH);
      digitalWrite(6,LOW); //avanza
   }
   else j--;
}

Serial.println();
```

```
   digitalWrite(5, HIGH);
   digitalWrite(5, LOW); //reseteo
  }

  Serial.println("-1"); Serial.flush();
  Serial.end();
  for(i=0;i<1;i++) i=-1;
}
```

Anexo E. Código del medidor de distancias Hamming

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>


#define N_claves 50
#define N_bits 13

int main()
{


    int i,j, t, c,aux=0,valor=0, cont[N_claves], maximo,maximo2, posicion,posicion2, distancias[N_claves];
    int med[N_claves][N_bits];
    FILE* medidas;
    FILE* resultados;
    FILE* distanciashamming;
    int clave[13]={1,1,0,1,0,1,0,1,0,1,1,0,0};

    medidas=fopen("Horizontal-50claves-13bits-inter-chip-1000cuentas.txt","r");
    resultados=fopen("resultados.txt","w");
    distanciashamming=fopen("distanciashamming.txt","w");

    if(medidas==NULL){printf("tolai");}

    for(i=0;i<N_claves;i++)
    {
        for(j=0;j<N_bits;j++)
        {

      fscanf(medidas,"%d",&med[i][j]);
    }
}


for(c=0;c<N_claves;c++)
{
    for(i=0;i<N_claves;i++)
{   valor=0;

    for(j=0;j<N_bits;j++)
    {

      aux=med[c][j]-med[i][j];
      if(aux!=0){valor++;}

    }//Nos saca el número de coincidencias que ha tenido la clave seleccionada a lo largo de todas las claves

    if(valor==0){cont[c]++;}

}

}

maximo=cont[0];
posicion=0;
for(c=1;c<N_claves;c++)
{
    if(cont[c]>maximo)
    {
```

```
            maximo=cont[c]; //Nos selecciona la clave que más coincidencias a tenido.
            posicion=c;
        }

    }

    //Ahora vamos a hacer que nos calcule la distancia hamming y nos de el máximo valor de la misma


        for(i=1;i<N_claves;i++)
    {   valor=0;

        for(j=0;j<N_bits;j++)
        {

          //aux=clave[j]-med[i][j];
          aux=med[posicion][j]-med[i][j];
          if(aux!=0){valor++;}


        }

       distancias[i]=valor;

    }

    maximo2=distancias[0];
    posicion2=0;
    for(c=1;c<N claves;c++)

  {
      if(distancias[c]>maximo2)
      {
          maximo2=distancias[c]; //Nos selecciona la clave que mayor distancia hamming tiene
          posicion2=c;
      }

  }

    int max=maximo2+1;

  //realiza histograma
  int hist[max];
  int hist1[max];

  for(c=0;c<max;c++)
  {
      hist1[c]=0;
      hist[c]=c;
  }

      for(i=0;i<N_claves;i++)
      {
        hist1[distancias[i]]++;
      }
```

```
fprintf(resultados,"Clave más repetida:\n");
for(j=0;j<N_bits;j++)
{
      fprintf(resultados,"%d",med[posicion][j]);
}
fprintf(resultados,"\n");
fprintf(resultados,"Clave con mayor distancia hamming:\n");
for(j=0;j<N_bits;j++)
{
      fprintf(resultados,"%d",med[posicion2][j]);
}

fprintf(resultados,"\n");
fprintf(resultados,"Mayor distancia hamming:\n");
fprintf(resultados,"%d",maximo2);

if(maximo2==0)
{

        fprintf(distanciashamming,"%d  %d\n",0,0);
        fprintf(distanciashamming,"%d  %d\n",hist[0],hist1[0]);

}else{
    for(j=0;j<max;j++)
{
      fprintf(distanciashamming,"%d  %d\n",hist[j],hist1[j]);
}
}

 fclose(medidas);
 fclose(resultados);
 fclose(distanciashamming);


 return 0;
}
```

Anexo F. Códigos de corrección de error (ECC)

F.1 Código Hamming (7,4)

En primer lugar, debemos entender lo que es un bit de paridad, dicho bit simplemente se incluye en la serie de datos y nos dice si el número de unos que hay en la misma es par o impar, siendo la convención que valga 1 cuando el número es impar y 0 si es par. De manera que en este código vamos a introducir de una manera proporcional a la longitud de la cadena de datos, un número de bits de paridad. Estos bits se colocaran en posiciones que sean potencia de 2 en la cadena (por ejemplo 1, 2, 4, 8, etc.). De manera que estos bits de paridad tomaran un valor determinado que nos de cómo paridad global de la cadena 0, pero condicionado con el hecho de que en función de su posición tendrá en cuenta solo unos determinados bits de datos, es decir, cada bit de paridad tomara en cuenta una cadena distinta extraída de la cadena inicial mediante la regla que se explica a continuación. La regla general es si el bit de paridad está en la posición n, salta n-1 bits y comprueba n bits, salta n bits y comprueba n bits… (Todo está referido a la posición que ocupan los bits dentro de la cadena de datos).

Una vez asignados los valores a los bits de paridad, enviamos los datos y suponemos que sufre un bit error (también lo puede sufrir el bit de paridad), es decir, un bit cambia. Con este código, al recibir dicha cadena con el bit cambiado, al volver a ver la paridad global de las distintas cadenas (referidas a los distintos bits de paridad), esta nos cambiaría en todas aquellas cadenas que tengan en cuenta el bit que ha cambiado, de forma que podemos saber cuál ha cambiado y corregirlo. Se llama en concreto (7,4) porque introduce tres bits de paridad a cuatro bits de datos y el problema principal es que solo puede corregir el error en un bit, si hay más de un error no corrige adecuadamente.

F.2 Repetición

Este código consiste en repetir el bit que se desea enviar una serie de veces, por ejemplo, si consideramos n=3, si tenemos un 1 enviaríamos "111". De forma que al sufrir esta cadena un bit error, obtendríamos una cadena con 2 unos en vez de 3. Este código da como bit final aquel que tenga más presencia en la cadena, es decir, si tenemos "101" nos dará un 1 y si tenemos "001" nos dará un 0. Por tanto, si enviamos un 1 mediante n=3, aunque sufriera un bit error, al seguir teniendo mayoría de unos obtendríamos un uno finalmente, corrigiendo el error. Conforme mayor sea n, más errores de bit podemos corregir, el problema está en que perdemos velocidad de transmisión de datos al tener que enviar un mismo dato n veces.

Estos son solo una serie de ejemplos de una vasta lista de ECC. A la hora de la elección de uno de ellos, se debe tener en cuenta su capacidad de corrección de datos y su complejidad, es decir, la dificultad de implementación, el consumo de energía, etc. Dichos ECC deben ser guardados en un servidor o NVM, que nos permita que cada vez que generemos un output con el fin de obtener una clave criptográfica, este no sufra variaciones.

Anexo G. Descripción y planificación de actividades

En este Anexo se presenta un resumen de las actividades realizadas y conocimientos adquiridos a lo largo del desarrollo del TFG, así como el número de horas destinadas para ello.

Tareas	Horas	Objetivos	Conocimientos y competencias
Physically Unclonable Function (PUF): Aplicaciones. Clasificación.	35	Estudio teórico y revisión bibliográfica.	• Definición de PUF. • Aplicaciones principales de las PUF. • Clasificación y propiedades inherentes de las PUF. • Ventajas y desventajas de las PUF.
Aprendizaje del lenguaje hardware Verilog. Aprendizaje del flujo de diseño en Vivado. Arquitectura FPGA.	55	Aprendizaje de las herramientas a utilizar a lo largo del TFG.	• Programación en Verilog. • Metodología de análisis, diseño e implementación en FPGA. • Aprendizaje de la arquitectura de las FPGAs. • Vivado.
Oscilador de anillo. PUF de osciladores de anillo.	20	Estudio teórico y revisión bibliográfica.	• Características de un oscilador de anillo. • Características de bloques digitales auxiliares. • Características de una RO-PUF completa. • Ventajas y desventajas de la RO-PUF.
Implementación software de los distintos módulos de la RO-PUF y de las herramientas de procesado de medidas.	50	Diseño y programación en Verilog y C de la PUF y las herramientas de procesado de medidas, respectivamente.	• Diseño, programación e implementación de las partes constituyentes de la RO-PUF, así como de las herramientas de procesado de medidas necesarias.
Realización de medidas	25	Trabajo en el laboratorio: Medición y análisis de los parámetros característicos de la PUF	• Comprobación de los parámetros característicos de la RO-PUF. • Procesamiento de los datos obtenidos. • Análisis de los datos obtenidos.
Elaboración de memoria. Preparación presentación.	15	Preparación y presentación del TFG.	• Extracción de resultados y conclusiones. • Competencias informacionales del grado.