Universidad Zaragoza

Trabajo Fin de Grado

Generador hardware de claves basado en funciones físicamente no clonables

Autor

Daniel Gil Marco

Directores

Carlos Sánchez Azqueta

Guillermo Díez Señorans

Facultad de Ciencias

Departamento de Ingeniería Electrónica y Comunicaciones

Junio 2020

Agradecimientos

En primer lugar, me gustaría dar las gracias a mis directores Guillermo Díez Señorans y Carlos Sánchez Azqueta por la atención y disponibilidad siempre que lo he necesitado. También a Santiago Celma Pueyo, por su colaboración y continuo seguimiento del trabajo, y a Miguel García Bosque, también por su continua disponibilidad y por la grandísima ayuda con sus diseños de VHDL para poder llevar a cabo el trabajo. Finalmente, agradecer a todo el departamento de Ingeniería Electrónica y Comunicaciones por su amabilidad y buen trato, y en especial a Óscar, el técnico de laboratorio, que siempre ha estado disponible para solucionar cualquier problema técnico que surgiera.

RESUMEN

GENERADOR HARDWARE DE CLAVES BASADO EN FUNCIONES FÍSICAMENTE NO CLONABLES

En este Trabajo Fin de Grado se ha llevado a cabo el estudio de las funciones físicamente no clonables (PUFs), profundizando en un caso concreto: las funciones físicamente no clonables basadas en osciladores de anillo de Galois, un tipo de oscilador de anillo diferente al oscilador de anillo simple.

Se ha realizado tanto un análisis teórico como un estudio experimental de dichos sistemas.

En la parte teórica se ha llevado a cabo un estudio de las PUFs, así como del mencionado oscilador de anillo de Galois. Además de esto, se ha estudiado también el flujo de diseño y la sintaxis necesaria para implementar físicamente las PUF en una matriz de puertas lógicas programable (FPGA), empleando lenguajes de descripción hardware tales como VHDL y verilog en el software Vivado.

Por otro lado, la parte experimental ha consistido en la toma de medidas en el laboratorio con las PUF ya implementadas en la FPGA, y el posterior tratamiento de datos y análisis de los resultados obtenidos.

LISTA DE ACRÓNIMOS

Acrónimo	Significado
PUF	Physical(ly) Unclonable Function
HDL	Hardware Description Language
VHDL	Combinación de VHSIC (Very High Speed Integrated Circuit) y HDL
FPGA	Field Programmable Gate Array
ID	Abreviatura de “Identification”
RFID	Radio Frequency Identification
RO	Ring Oscillator
FIRO	Fibonacci Ring Oscillator
GARO	Galois Ring Oscillator
RO-PUF	Ring Oscillator- Physical(ly) Unclonable Function
GARO-PUF	Galois Ring Oscillator- Physical(ly) Unclonable Function
LFSR	Linear Feedback Shift Register
LUT	Look Up Table
CLB	Configurable Logic Block
RTL	Register-Transfer Level
HD	Hamming Distance
FAR	False Acceptance Rate
FRR	False Rejection Rate

LISTA DE FIGURAS

Figura 1: Arquitectura de una GARO-PUF 5
Figura 2: Esquema de un oscilador de anillo de Galois formado por n inversores 6
Figura 3: Configurable Logic Block de la FPGA 7
Figura 4: Diagrama de flujo de diseño de Vivado 8
Figura 5: Estructura del oscilador de anillo de Galois 10
Figura 6: Implementación de un oscilador de anillo en la FPGA 12
Figura 7: Ejemplo de solapamiento entre las curvas de intra-distancia e inter-distancia 17
Figura 8: Valor medio de la Distancia Hamming en la GARO-PUF para diferentes temperaturas y 20 claves 19
Figura 9: Varianza de las Distancias Hamming en la GARO-PUF para diferentes temperaturas y 20 claves 20
Figura 10: Valor medio de la Distancia Hamming en la RO-PUF para diferentes temperaturas y 20 claves 21
Figura 11: Varianza de las Distancias Hamming en la RO-PUF para diferentes temperaturas y 20 claves 21
Figura 12: Valor medio de la Distancia Hamming en la RO-PUF para diferentes temperaturas y 100 claves 22
Figura 13: Varianza de las Distancias Hamming en la RO-PUF para diferentes temperaturas y 100 claves 22
Figura 14: Ajuste a una distribución binomial de las intra-distancias y las inter-distancias para la placa 1 24

LISTA DE TABLAS

Tabla 1: Medidas en la GARO-PUF con 20 claves 19
Tabla 2: Medidas en la RO-PUF con 20 claves 20
Tabla 3: Medidas en la RO-PUF con 100 claves 22

INDICE

1. Introducción 1

1.1 Objetivos del trabajo 2

1.2 Metodología 2

1.3 Herramientas utilizadas 2

2. Funciones físicas no clonables 2

3. Oscilador de anillo de Galois 4

4. Metodología de implementación 6

4.1 Arquitectura de la FPGA 6

4.2 Flujo de diseño en Vivado e implementación en la FPGA 7

5. Diseño e implementación de la PUF de osciladores de anillo de Galois 8

6. Conceptos clave 13

6.1 Distancia Hamming 13

6.2 Distribución Binomial 14

6.3 Intra-distancia e inter-distancia 15

7. Identificación y autenticación de una entidad basada en PUF 15

8. Caracterización experimental 18

8.1 Medida de la intra-distancia de una GARO-PUF en una FPGA para distintas temperaturas 18

8.2 Medida de la intra-distancia de una RO-PUF en una FPGA para distintas temperaturas 20

8.3 Medida de la inter-distancia 23

8.4 Identificación y autenticación 23

9. Conclusión 24

10. Bibliografía 25

ANEXO A: Códigos VHDL para el diseño de la PUF en Vivado I

A.1. Módulo Clock Divider I

A.2. Módulo de selección de osciladores II

A.3. Módulo suma VII

A.4. Módulo comparador VIII

A.5. Módulo Control IX

ANEXO B: Códigos en Arduino para la generación de claves XI

B.1. Oscilador de anillo de Galois XI

B.2. Oscilador de anillo simple XIII

ANEXO C: Código en C para el cálculo de Distancias Hamming y estadística XV

1. Introducción

Este trabajo trata sobre las denominadas funciones físicas no clonables, (PUFs), concretamente sobre las funciones físicas no clonables de oscilador de anillo de Galois. Los osciladores de anillo de Galois son unos sistemas mucho más complejos que los osciladores de anillo normales. Además de esto, son unos sistemas muy novedosos y por lo tanto poco estudiados.

Las PUFs generan una salida que es única en cada dispositivo, por lo que puede hacerse una analogía entre esta salida y una "huella dactilar" electrónica. Las PUFs pueden emplearse como generador de claves, que va a ser el caso de este trabajo.

Los osciladores de anillo son una sucesión impar de inversores (y en el caso de los osciladores de Galois también de puertas lógicas) realimentada de tal modo que se crea un sistema inestable cuya salida oscila constantemente entre los valores 0 y 1 lógicos. El uso de osciladores de anillo para construir una PUF es debido a su utilidad para amplificar arbitrariamente diferencias, que a priori son ínfimas, en los tiempos de transmisión de dos circuitos idénticos.

Para compensar la poca robustez de una PUF de oscilador de anillo simple frente a variaciones de temperatura [1], en este trabajo se quiere comprobar si, empleando una PUF de oscilador de anillo de Galois en lugar de osciladores de anillo simples, la robustez de la PUF se incrementa y la temperatura deja de ser un factor relevante. El proceso de medida que se va a llevar a cabo consiste en la generación de claves formadas por un determinado número de dígitos, que serán 0 o 1. Estas claves se generan mediante la comparación por parejas de un determinado número de osciladores de anillo de Galois que constituyen la PUF.

En teoría, una misma PUF implementada en un chip concreto debería generar siempre la misma clave, es decir, la misma secuencia de 0's y 1's. Sin embargo, esto no es así [2], y dependiendo de la robustez de la PUF, la variabilidad en las claves será mayor o menor: si la PUF es muy robusta, habrá pocos dígitos que diferirán de una clave a otra; en cambio si no es muy robusta, habrá muchas variaciones entre las claves. Como es lógico, lo que se busca es una PUF muy robusta que genere siempre la misma clave, ya que, de no ser así, un sistema podría ser atacado con facilidad [3]. En el caso que concierne a este trabajo, por ejemplo, dada una PUF de oscilador de anillo que genera una determinada clave, si al variar la temperatura de una forma bastante significativa la PUF pierde robustez, la clave perdería toda su seguridad.

Por tanto, como se ha dicho, se procederá a comprobar si existe la buscada robustez en las PUF de oscilador de anillo de Galois al variar la temperatura. Para poder comparar los resultados obtenidos, se llevará a cabo el mismo proceso con una PUF de oscilador de anillo simple.

Una de las características de las PUFs, es que generan una salida distinta y particular según el dispositivo en el que esté implementada. Para entenderlo mejor, la PUF es siempre la misma, es decir, el oscilador de anillo tiene los mismos componentes independientemente del dispositivo en que se implementa. Sin embargo, en cada uno de los casos, se obtendrá una clave distinta.

Este trabajo también busca analizar este comportamiento. Implementando una misma PUF de oscilador de anillo de Galois en diferentes dispositivos, se procederá a la generación de claves al igual que en la primera parte del trabajo, con la diferencia de que, en lugar de estudiar las diferencias entre las claves generadas por la PUF en el mismo dispositivo, se compararan las claves correspondientes a los distintos dispositivos.

En un Trabajo Fin de Grado previo (Rubén Martín Pinardel, junio 2019, *Diseño CMOS de funciones físicas no clonables)*, se llevó a cabo un estudio sobre las PUFs de oscilador de anillo simple. En este Trabajo Fin de Grado, en lugar de emplear osciladores de anillo simple, se utilizan los osciladores de anillo de Galois, dándole continuidad al estudio anterior.

1.1 Objetivos del trabajo

Los objetivos de este Trabajo Fin de Grado son una introducción al concepto de PUF, el conocimiento del lenguaje de descripción hardware y su uso para implementar las PUFs en FPGAs y el análisis de resultados y extracción de conclusiones tras dicha implementación.

1.2 Metodología

El primer paso de este trabajo ha sido realizar un estudio teórico previo sobre las PUFs, ya que es un concepto relativamente nuevo y que no está incluido en los estudios del Grado en Física. Posteriormente se ha estudiado el oscilador de anillo de Galois, continuando con la descripción del material y el software empleado. Se han visto unos conceptos matemáticos clave y finalmente se ha realizado el análisis de las medidas y se han extraído las principales conclusiones.

1.3 Herramientas utilizadas

- **Vivado:** Software producido por Xilinx para síntesis y análisis de diseños HDL, con características que permiten el desarrollo de sistemas en FPGAs y síntesis de alto nivel.
- **VHDL:** Lenguaje utilizado para describir circuitos digitales y para la automatización de diseño electrónico. Con el software Vivado, se emplea generalmente para programar FPGAs.
- **Verilog:** Lenguaje de descripción hardware (HDL) empleado en Vivado para implementar circuitos en las FPGA.
- **Placas PYNQ-Z2:** Placas desarrolladas por Zynq. Cada una de estas placas posee una FPGA xc7z020-1CLG400C donde se implementan los osciladores de anillo. Como puerto de entrada se utiliza un puerto micro USB, y los puertos de salida se conectan a los pines de una placa Arduino UNO.
- **Arduino UNO:** Placa desarrollada por ARDUINO, programada y alimentada por cable USB y a cuyos pines analógicos se conecta la placa PYNQ.
- **Cámara térmica:** Cámara en cuyo interior se colocan las placas y que permite alcanzar la temperatura deseada. Se trata del modelo Fitoterm 22E, su rango de temperatura va de -40°C a +160°C y se alimenta eléctricamente con 230 V, 50 Hz y 16 A.

2. Funciones físicas no clonables

Una función física no clonable [2] (PUF, en inglés Physical Unclonable Function) es una entidad que utiliza la variabilidad en el proceso de fabricación para generar una salida específica de un dispositivo, que generalmente es proporcionada como un número binario. Se puede considerar esta salida como la “huella dactilar” de un dispositivo, ya que, en teoría, es única. Se dice en teoría porque las PUFs normalmente no proporcionan exactamente siempre la misma salida,

muestran errores de bits. Estos errores pueden ser tanto aleatorios como sistemáticos. Los errores aleatorios son generados por el ruido del circuito, mientras que los sistemáticos se generan por falta de coincidencia entre los parámetros de los componentes involucrados.

Una PUF está hecha de varios componentes definidos por variaciones de parámetros locales. Las diferencias entre los componentes son llamados desajustes locales. Dependiendo del enfoque de la PUF, estos parámetros locales son combinados, comparados o leídos directamente para generar la salida binaria.

Tras esta breve introducción, se va a profundizar en la terminología de las palabras que forman el acrónimo PUF:

- Physical(ly): en un principio, se hablaba de "función física no clonable", pero más tarde comenzó a usarse la terminología "función físicamente no clonable". Ambos términos se refieren al mismo concepto, pero estrictamente hay una diferencia. El primer caso se refiere a una función física que no es clonable, mientras que el segundo es una función que no es clonable físicamente. La segunda terminología se ajusta más a la realidad [4], ya que normalmente muchas PUFs no son clonables físicamente pero sí lo son matemáticamente.
- Unclonable: dado que la variación de los componentes de una PUF no puede controlarse desde el exterior, una PUF no puede ser replicada. Esto es lo que la hace no clonable. Este adjetivo es el que refleja la propiedad distintiva de una PUF.
- Function: dependiendo de la aplicación, la salida de la PUF depende de una señal de entrada, por lo que se puede considerar la PUF como una función. Sin embargo, en el puro sentido matemático una PUF no es estrictamente una función, ya que una sola entrada puede estar relacionada con más de una salida debido a los efectos incontrolables del entorno físico y el ruido aleatorio en la generación de respuesta. Por lo tanto, es más correcto hablar de una función probabilística, es decir, una función para la cual parte de la entrada es una variable aleatoria incontrolable.

Es interesante clasificar los distintos tipos de PUFs según distintos criterios:

1. Una posible clasificación se basa en la naturaleza electrónica de sus características de identificación:

- PUFs no electrónicas: con no electrónico se refiere al origen del comportamiento de la PUF, no la forma en que se procesan o almacenan las respuestas de la PUF, que a menudo usa electrónica y circuitos digitales.
- PUFs electrónicas: se trata de circuitos electrónicos integrados que exhiben un comportamiento de PUF.
    - PUFs de silicicio: una subclase de las anteriores. Los circuitos electrónicos están integrados en un chip de silicio.

2. Una clasificación importante de las PUFs en función de sus propiedades de construcción es el de las PUFs intrínsecas. Se considera que una PUF debe cumplir dos condiciones para ser llamada PUF intrínseca:

- Sus evaluaciones se realizan internamente por equipos de medición integrados.
- Sus características aleatorias específicas se introducen implícitamente durante su proceso de producción.

3. Se puede realizar una última clasificación de PUFs basada explícitamente en las propiedades de seguridad de su comportamiento desafío-respuesta. (Con la terminología desafío -

respuesta se hace referencia a la relación entrada (input) – salida (output) de la PUF). Una PUF se llama fuerte si, incluso después de dar acceso a un adversario a una instancia de la PUF durante un periodo prolongado de tiempo, aún es posible presentar un desafío para el que con alta probabilidad el adversario no conoce la respuesta.

Una vez visto qué son las PUFs y cómo pueden clasificarse, es interesante discutir su utilidad, para qué sirven. Existe una amplia gama de aplicaciones entre las que se puede citar el comercio, donde las PUFs forman parte integrada de las etiquetas RFID; en la televisión de pago, para que solo los usuarios autorizados puedan ver el programa; para evitar el negocio ilegal de la venta de chips (night-shift problem); para evitar la falsificación; y en la protección del código FPGA. Las PUFs se introducen para reducir costos, aumentar el nivel de seguridad o, a veces, ambos.

Finalmente, para terminar con esta introducción teórica sobre las PUFs se van a comentar tres de sus aplicaciones básicas: identificación, autenticación y generación de claves. Dado que la autenticación electrónica se realiza con la ayuda de herramientas criptográficas, las aplicaciones básicas se pueden agrupar en aplicaciones con propósitos criptográficos y no criptográficos.

- Identificación: Se asigna una ID conocida a una entidad desconocida. En el contexto de microelectrónica y PUFs, dicha ID se asigna a un chip. El requisito básico para la identificación es que para cada entidad se asigne una ID única.
- Autenticación: Es el procedimiento para verificar una identificación reclamada de una entidad. También es una generalización de la identificación, ya que, si una entidad es autentificable, es identificable.
- Generación de claves: Una PUF puede usarse para generar claves con propósitos criptográficos, para lo cual es esencial que la clave generada sea siempre la misma para la misma PUF. Dado que las PUFs siempre producen errores de bits en su salida, se debe establecer un mecanismo de corrección de errores. Por lo tanto, la corrección de errores es uno de los temas principales sobre PUFs.

3. Oscilador de anillo de Galois

El oscilador de anillo es una sucesión impar de inversores realimentada, de tal modo que se crea un sistema astable cuya salida oscila constantemente entre los valores 0 y 1 lógicos.

En 2006, Jovan Golić [5,6] propuso un nuevo método para la generación de números aleatorios verdaderos utilizando nuevas estructuras llamadas osciladores de anillo de Fibonacci y osciladores de anillo de Galois (FIRO y GARO respectivamente). Estas estructuras están basadas en la estructura de los osciladores de anillo, pero, en lugar de usar una realimentación circular simple, utilizan una realimentación más compleja incorporando puertas XOR en una forma análoga a las configuraciones de Fibonacci y Galois de una LFSR [5,6], que, traducido como "registro de desplazamiento con realimentación lineal", se trata de un registro de desplazamiento en el cual la entrada es un bit proveniente de aplicar una función de transformación lineal a un estado anterior. Este trabajo se centra en el oscilador de anillo de Galois.

La idea de esta propuesta es combinar las propiedades pseudo-aleatorias de las LFSRs, con las propiedades aleatorias verdaderas de los osciladores de anillo debido al ruido en fase y frecuencia (jitter).

Las conexiones de realimentación se especifican con coeficientes f_i y, por lo tanto, la configuración se puede definir utilizando un polinomio binario, $f(x)=\sum_{i=0}^{r} f_i x^i$, $f_0=f_r=1$ llamado polinomio de realimentación. Si $f_i=1$, el interruptor correspondiente está cerrado mientras que, si $f_i=0$, el interruptor correspondiente está abierto (en ese caso, la XOR no está presente). Dado un polinomio de realimentación de orden r, es posible implementarlo fácilmente en una FPGA utilizando exactamente r LUTs. Para hacer esto, si $f_i=1$, la función implementada en la i-ésima LUT es una operación XNOR mientras que, si $f_i=0$, la función implementada en la i-ésima LUT es una operación NOT.

El retraso de propagación de las LUT no depende de la función implementada [7], por lo que el retraso de tiempo total solo dependerá del polinomio primitivo, haciendo el estudio de estos sistemas más fácil.

Dado que estas estructuras presentan algunos problemas para ser utilizadas para generar números aleatorios verdaderos, se propone una nueva aplicación para estos sistemas: utilizar las diferencias estadísticas de las secuencias generadas por distintas instancias de un mismo oscilador para construir una PUF. Se define pues el sesgo de un oscilador como el número de 0's y 1's que tiene la secuencia generada por el oscilador.

La estructura básica de una GARO-PUF consiste en una matriz de k osciladores, cada uno de ellos muestreados con un registro, generando una secuencia binaria. Se utilizan un par de multiplexores para seleccionar qué osciladores comparar y se mide el sesgo de cada uno. Finalmente, usando un bloque de comparación, si el número de 1's del primer oscilador es mayor que el del segundo, la salida es 1. De lo contrario, la salida es 0 (a diferencia de una PUF de oscilador de anillo simple donde lo que se comparan son las frecuencias de los osciladores, siendo la salida 1 cuando el primer oscilador tiene una frecuencia mayor, y siendo 0 en el caso contrario). Un esquema que resume la estructura de una PUF de osciladores de anillo de Galois se muestra en la Figura 1, mientras que en la Figura 2 aparece la estructura de un oscilador de anillo de Galois.

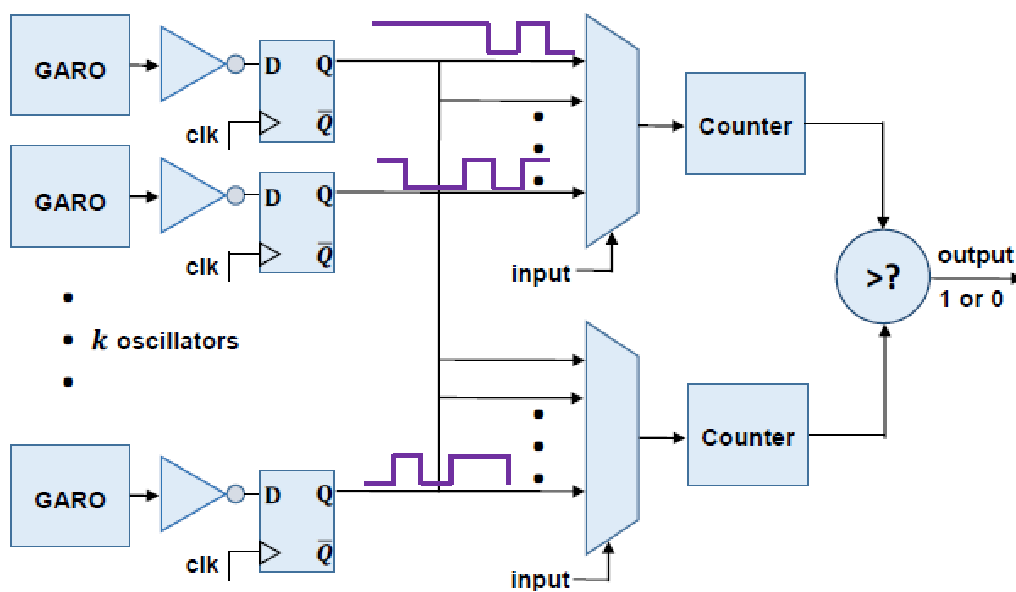

Figura 1: Arquitectura de una GARO-PUF

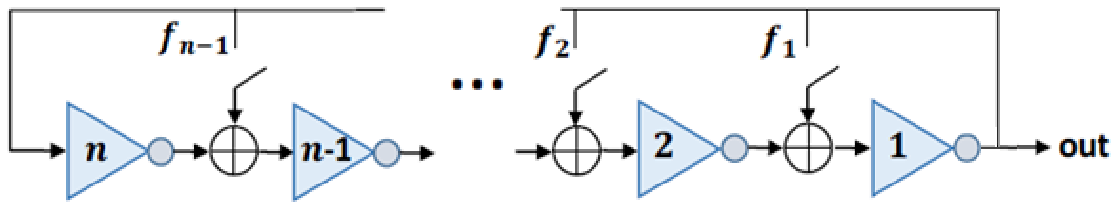

Figura 2: Esquema de un oscilador de anillo de Galois formado por n inversores

Un oscilador de anillo de Galois consiste en *n* inversores conectados en cascada de tal forma que la salida de cada inversor forma la entrada del siguiente, y la salida del último inversor define directamente la señal de realimentación. Las conexiones de realimentación están especificadas por los coeficientes binarios f_i con la convención de que el interruptor correspondiente está cerrado si f_i=1 y abierto si f_i=0, en cuyo caso la correspondiente puerta XOR no está presente, como se ha mencionado anteriormente. Los coeficientes de realimentación y los correspondientes inversores están indexados de derecha a izquierda. La entrada del primer inversor en la cascada está directamente definida por la señal de realimentación. Si f_i=0, la entrada del inversor *i*-ésimo está directamente definida por la salida del inversor (*i*+1)-ésimo. Si f_i=1, la entrada del inversor *i*-ésimo está formada por el resultado del XOR entre la salida del inversor (*i*+1)-ésimo con la señal de realimentación. En la Figura 2, la señal de salida proviene del último inversor.

En un oscilador de anillo de Galois se cumple el siguiente teorema [5]:

Teorema: *Un oscilador de anillo de Galois no tiene un punto fijo sí y solo sí:*

$$f(1) = 0 \text{ y } r \text{ es impar}$$

Esta condición significa que $f(x)=(1+x)h(x)$, siendo $h(x)$ un polinomio primitivo, esto es, un polinomio irreducible con un periodo máximo 2^{r-1}-1, y que el grado de $f(x)$ es impar.

Se recomienda un polinomio primitivo $h(x)$ que asegure buenas propiedades pseudoaleatorias de la señal de salida. Un número pseudoaleatorio es un número generado en un proceso que parece producir números al azar, pero no lo hace realmente, ya que su generación parte de algoritmos determinísticos, lo cual significa que se obtendrá siempre el mismo resultado bajo las mismas condiciones iniciales. Un número verdaderamente aleatorio es aquel que es generado mediante un proceso no determinista, usando algún tipo de fuente de ruido presente en la naturaleza. La generación de claves con PUFs se trata pues de un proceso pseudoaleatorio que debería generar siempre la misma clave, sin embargo, esto no es así, y de ahí el estudio más a fondo que se va a realizar en este trabajo. La generación de números pseudoaleatorios es muy importante en criptografía, con la cual están estrechamente relacionadas las PUFs.

4. Metodología de implementación

4.1 Arquitectura de la FPGA

Una matriz de puertas lógicas programable o FPGA [8] es un dispositivo programable que contiene bloques digitales cuya interconexión y funcionalidad puede ser configurada por software, mediante un lenguaje de descripción hardware especializado.

El diseño en FPGA no está ideado para construcciones combinacionales como los osciladores de anillo, y el software optimizador de la implementación eliminará redundancias como una cadena de inversores (que puede sustituirse lógicamente por un único inversor si la cadena es impar, o por un simple cable si la cadena es par). Por ello, debe descenderse al nivel físico de la implementación y ordenar (a través de las restricciones adecuadas) a Vivado que respete la construcción del oscilador.

La FPGA está compuesta por CLBs. Cada uno de estos elementos consta de un par de celdas independientes, cada una de las cuales está compuesta por cuatro LUTs (tablas de verdad) con capacidad para 6 entradas y 8 elementos de almacenamiento [9]. Las LUT son unas memorias (programables) que almacenan una función lógica tabulada, es decir, producen una salida en función de unas entradas dadas. Vivado permite programar cada LUT individual y concretamente (una LUT determinada en la ubicación física elegida de la placa). El conjunto de CLB conforma una matriz que toma las siguientes coordenadas espaciales: en la coordenada x, desde 0 hasta 43, y en la coordenada y, desde 0 hasta 99. En la Figura 3 se muestra uno de los CLBs que constituyen la FPGA donde se pueden ver sus diferentes recursos. Las LUTs corresponden a los 4 elementos de la izquierda.

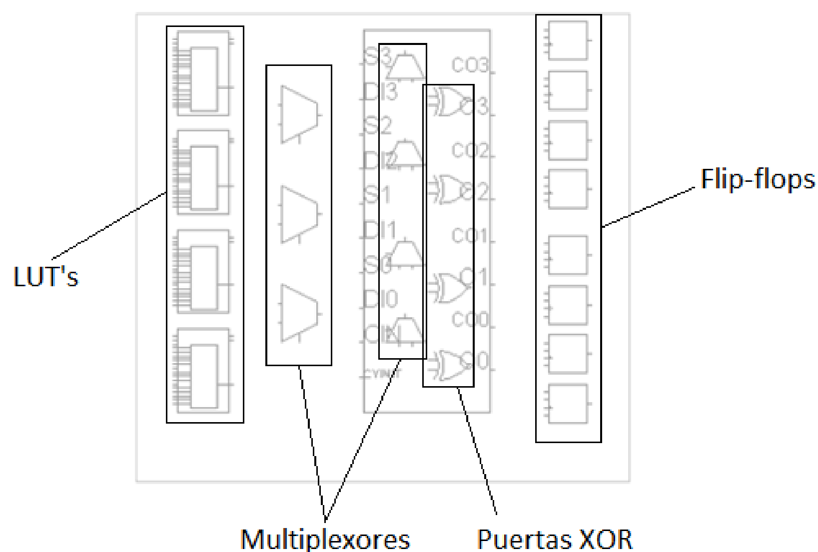

Figura 3: Configurable Logic Block de la FPGA

4.2 Flujo de diseño en Vivado e implementación en la FPGA

Para implementar un circuito en una FPGA, en este caso en concreto la PUF de oscilador de anillo de Galois, se emplea el software de diseño Vivado, con los circuitos en sí descritos en lenguaje VHDL.

Una vez abierto Vivado, el primer paso es crear un proyecto y seleccionar la FPGA en la que se va a implementar la PUF. Tras haber configurado el proyecto, hay que importar el circuito en formato verilog o VHDL y las restricciones impuestas físicamente a la FPGA, principalmente la configuración de puertos físicos.

A continuación, ya se puede cargar el programa en la FPGA, para lo cual se seguirán unos pasos que constituyen el flujo de diseño de Vivado. El papel de este flujo de trabajo es convertir el archivo que está en lenguaje HDL, en un circuito. El diagrama de flujo es el que se muestra en la Figura 4.

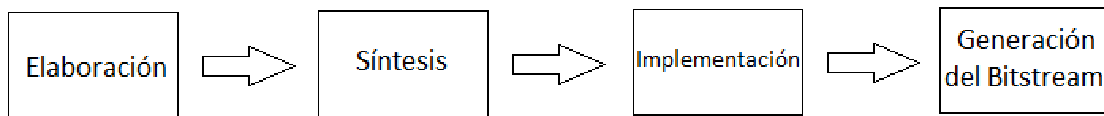

Figura 4: Diagrama de flujo de diseño de Vivado

Los pasos que constituyen el flujo son los siguientes:

1. Elaboración: Se convierte la descripción RTL, es decir, el código escrito, en un circuito equivalente constituido por elementos lógicos ideales.

2. Síntesis: Se sustituyen los elementos del circuito por las descripciones de elementos lógicos reales. En el diseño de FPGAs se trata de realizaciones que Vivado es capaz de programar en los CLBs.

3. Implementación: Se pasa de una descripción lógica en formato VHDL a una descripción física del circuito sobre la placa. Cada módulo es colocado en un lugar físico de la placa y se realizan las conexiones eléctricas.

4. Generación del bitstream: Una vez se ha terminado el diseño, se codifica para que sea compatible con la placa y se programe correctamente. Este paso solo ocurre en el diseño de FPGAs.

Una vez se han realizado estos pasos de diseño, ya se puede cargar el archivo generado (bitstream) en la FPGA desde el ordenador a través de un puerto micro USB.

5. Diseño e implementación de la PUF de osciladores de anillo de Galois

Una vez visto el flujo de diseño en Vivado, se particulariza al diseño del oscilador de anillo.

Se indicó en el apartado 4.1 “Arquitectura de la FPGA” que Vivado permite programar la tabla de verdad de cada LUT individual y concretamente. Para el oscilador, se utiliza LUT1 (una única entrada) para los inversores y LUT2 (dos entradas) para las puertas lógicas XNOR. Aunque en el apartado 3 “Oscilador de anillo de Galois” se habla de puertas XOR, en diseño digital CMOS, las funciones lógicas se implementan negadas por defecto. Si no se implementan negadas, se requiere de un inversor extra y por eso se tiende a evitar. La lógica al final es la misma.

Estas LUT individuales tienen que tener una ubicación restringida para colocarlas físicamente en lugares concretos de la placa, para lo cual se utilizan tres variables:

1. BEL (Basic Element): especifica la LUT concreta que debe programarse para constituir un módulo dado dentro de una celda concreta. Cada celda tiene cuatro LUTs llamadas A, B, C y D. Estas LUTs son independientes. A la variable BEL se le asigna uno de estos cuatro valores, por ejemplo <<A6LUT>>, que indica que el módulo al que se le añade la propiedad BEL debe programarse en la LUT A, y que dicha LUT tendrá 6 posibles entradas.

2. LOC: especifica la celda de la matriz que contiene la placa donde se situará la LUT. La sintaxis de esta variable es LOC = "SLICE_X_iY_j" siendo i y j la posición de la celda en la matriz de bloques lógicos de la FPGA ($0 \leq i \leq 43$, $0 \leq j \leq 99$).

3. DONT_TOUCH: sirve para impedir que el programa elimine los módulos.

Se ve en el siguiente código la aplicación de las variables a una LUT concreta, esto se tiene que repetir para todas las LUT que se vayan a utilizar:

```
attribute LOC of node_1_inverter  : label is "SLICE_X" & integer'image(integer(XOS)) & "Y" & integer'image(integer(YOS));
attribute BEL of node_1_inverter : label is  "A6LUT";
attribute keep_hierarchy of node_1_inverter : label is "yes";
attribute dont_touch of node_1_inverter : label is "yes";
```

El siguiente paso, una vez se han fijado las LUTs, es definir las componentes que conforman cada uno de los 100 osciladores por los que está compuesta la PUF de este trabajo. El primer caso, LUT1, corresponde a un inversor, y el segundo, LUT2, a una puerta XNOR.

```
component LUT1 is
    generic (
        INIT: bit_vector(0 to 1) := (others => '0')
    );
    port (
        O : out std_logic;
        I0 : in std_logic
    );
end component;

component LUT2 is
    generic(
      INIT : bit_vector(0 to 3) := (others => '0')
    );
    port(
        O : out std_ulogic;
        I0 : in std_ulogic;
        I1 : in std_ulogic
    );
end component;
```

Además de esto también se define una componente distinta para un registro de muestreo:

```
component FDRE is
    generic(
        INIT : bit := '0';
        IS_C_INVERTED : bit := '0';
        IS_D_INVERTED : bit := '0';
        IS_R_INVERTED : bit := '0'
    );
```

```
    port(
        Q    : out std_ulogic := TO_X01(INIT);
        C    : in std_ulogic;
        CE   : in std_ulogic;
        D    : in std_ulogic;
        R    : in std_ulogic
    );
end component;
```

Para este registro de muestreo no hay que fijar una LUT, sino otro componente de los CLBs llamado flip-flop. Sí que se fijará otra LUT para un inversor adicional de salida.

Posteriormente, con las componentes definidas, se crea el oscilador. La estructura que tiene el oscilador de anillo en este caso se muestra en la Figura 5.

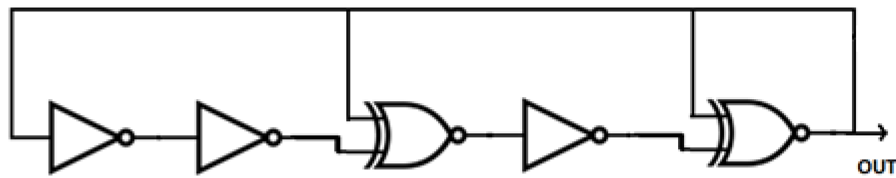

Figura 5: Estructura del oscilador de anillo de Galois

Cada uno de estos componentes va en una de las LUT que se han fijado previamente. Para colocar un inversor en una LUT, en este primer caso en la que se ha puesto como ejemplo al principio de esta explicación, el código es:

```
node_1_inverter: component LUT1
generic map (
    INIT(0) => '0',
    INIT(1) => '1'
)
port map (
    O  => node_output(0),
    I0 => node_output(4)
);
```

Y para colocar una puerta XNOR, en este caso en lo que sería la tercera LUT:

```
node_3_xnor: component LUT2
generic map (
    INIT(0) => '1',
    INIT(1) => '0',
    INIT(2) => '0',
    INIT(3) => '1'
)
port map(
    O  => node_output(2),
    I0 => node_output(4),
    I1 => node_output(1)
);
```

El inversor de salida y el registro de muestreo se colocan de la misma forma, en otras LUT:

```
output_inverter: component LUT1
generic map (
    INIT(0) => '0',
    INIT(1) => '1'
)
port map (
    O  => random_analog_signal,
    I0 => node_output(4)
);
```

```
FF: component FDRE
generic map (
      INIT => '0',
      IS_C_INVERTED => '0',
      IS_D_INVERTED => '0',
      IS_R_INVERTED=> '0'
      )
port map(
    Q   => random_bit,
    C   => sampling_clk,
    CE  => '1',
    D  => random_analog_signal,
    R   => '0'
);
```

Este proceso hay que hacerlo para cada oscilador, cada uno en un archivo distinto. Esto se indica al principio de cada código de oscilador:

```
entity Golic5_X0_Y0 is
    Port (
        master_clk : in STD_LOGIC;
        sampling_clk    : in STD_LOGIC;
        reset : in STD_LOGIC;
        random_bit : out STD_LOGIC
    );
end Golic5_X0_Y0;
```

De esta forma, cada oscilador está diseñado como un módulo en Vivado, y luego puede ser invocado en el módulo principal de la siguiente manera:

```
component Golic5_X0_Y0 is
    Port (
        master_clk : in STD_LOGIC;
        sampling_clk : in STD_LOGIC;
        random_bit : out STD_LOGIC
    );
end component;
```

```
myGolic5_X0_Y0: component Golic5_X0_Y0
                Port map(
                    master_clk => master_clk,
                    sampling_clk => Clk_100kHz_buffer,
                    random_bit => array_of_bits_buffer(0)
                );
```

El primer fragmento se escribe como componente de la arquitectura del módulo principal y el segundo fragmento está dentro ya del propio código y utiliza la definición previa para diseñar la PUF completa.

Todo lo explicado hasta ahora hay que realizarlo para los 100 osciladores. En la Figura 6 se muestra un oscilador implementado en la FPGA, visto desde Vivado. Cada uno de los componentes en azul de los CLBs corresponden a las LUT fijadas para los inversores, y el componente pequeño del CLB de la derecha corresponde al flip-flop utilizado para el registro de muestreo. Estos componentes están conectados entre sí mediante cables.

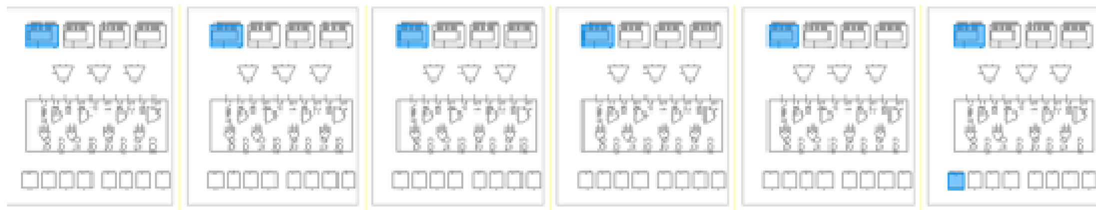

Figura 6: Implementación de un oscilador de anillo en la FPGA

Aparte de los módulos en los que se diseña cada uno de los osciladores de anillo, hay más módulos que están incluidos dentro del módulo principal, los cuales van a ser explicados a continuación, y su código se añadirá como anexo debido a su larga extensión:

- Clock Divider (Anexo A.1): Recibe como entrada la señal de reloj principal de 125 MHz y saca como salida otras señales de reloj de menor frecuencia. Estas señales de reloj son necesarias para indicar cuando se deben realizar ciertas acciones como muestrear los bits, realizar las comparaciones o transmitir los bits.
- Módulo de selección de osciladores (Anexo A.2): La PUF cuenta con 100 osciladores de anillo de Galois. Este módulo se encarga de seleccionar los osciladores por parejas, para realizar su posterior comparación. Al haber 100 osciladores y ser seleccionados de dos en dos de manera adyacente (primero con segundo, segundo con tercero...), se obtiene un total de 99 selecciones, que es el número de dígitos que tiene cada clave, ya que cada uno de los dígitos se obtiene tras la comparación de cada pareja de osciladores.
- Módulo suma (Anexo A.3): Una vez seleccionados los osciladores, hay que medir el sesgo de cada oscilador. Este módulo se encarga de contar el número de 1's de cada secuencia.
- Módulo comparador (Anexo A.4): Se comparan los resultados obtenidos por los anteriores módulos. Si el número de 1's del primer oscilador es mayor que el del segundo, el dígito de salida será 1, y en caso contrario será 0. Tras realizar las 99 comparaciones, se obtienen los 99 dígitos que componen cada clave.
- Control (Anexo A.5): Se encarga de que los módulos se comuniquen entre ellos correctamente. Este módulo recibe entradas de unos módulos y manda entradas a otros.

El uso de los módulos suma y comparador se debe a que con la PUF de oscilador de anillo de Galois el procedimiento es distinto a una PUF de osciladores de anillo simple. En este caso, simplemente se seleccionarían los osciladores por parejas y se compararía su frecuencia de oscilación.

Para importar el circuito a Vivado, hay que añadir el módulo principal, el cual incluye a todos los demás. Una vez importado el circuito, quedan por añadir las restricciones.

Los archivos de restricciones sirven para influir en la realización del circuito sobre la superficie de la placa. En este caso, las restricciones van a consistir en la configuración de pines de entrada/salida. Cada placa posee un archivo maestro que contiene todos los puertos. En este archivo hay que descomentar las líneas correspondientes a los pines que se van a utilizar y añadir este archivo, el cual está en formato XDC (xilinx design constraint), a la base de datos del proyecto de Vivado. A continuación, se muestran las restricciones (puertos) utilizadas para la PUF de oscilador de anillo de Galois.

```
## Clock signal 125 MHz

set_property -dict { PACKAGE_PIN H16   IOSTANDARD LVCMOS33 } [get_ports {clock }]; #IO_L13P_T2_MRCC_35 Sch=sysclk
create_clock -add -name sys_clk_pin -period 8.00 -waveform {0 4} [get_ports { clock }];

##Arduino Digital I/O On Outer Analog Header
##NOTE: These pins should be used when using the analog header signals A0-A5 as digital I/O
###pines PARES: entradas fpga->arduino; IMPARES: salidas fpga<-arduino

set_property -dict { PACKAGE_PIN Y11   IOSTANDARD LVCMOS33 } [get_ports { out_comp }]; #IO_L18N_T2_13 Sch=a[0]
set_property -dict { PACKAGE_PIN Y12   IOSTANDARD LVCMOS33 } [get_ports { rst_global }]; #IO_L20P_T3_13 Sch=a[1]
set_property -dict { PACKAGE_PIN W11   IOSTANDARD LVCMOS33 } [get_ports { out_igual }]; #IO_L18P_T2_13 Sch=a[2]
set_property -dict { PACKAGE_PIN V11   IOSTANDARD LVCMOS33 } [get_ports { arduino }]; #IO_L21P_T3_DQS_13 Sch=a[3]
set_property -dict { PACKAGE_PIN T5    IOSTANDARD LVCMOS33 } [get_ports { fin }]; #IO_L19P_T3_13 Sch=a[4]
```

6. Conceptos clave

6.1 Distancia Hamming

La Distancia Hamming (HD), es un parámetro de la teoría de la información llamado así en honor al matemático Richard W. Hamming.

Dadas dos cadenas binarias de la misma longitud, se define su Distancia Hamming como el número de bits en los que difiere una de otra. A continuación, se muestra un ejemplo:

Dadas dos cadenas como las siguientes:

Cadena 1: 0 0 0 1 0 1 1 0

Cadena 2: 1 1 0 1 1 0 1 0

Al hacer una XOR de los bits de las dos cadenas,

XOR: 1 1 0 0 1 1 0 0

se puede determinar la HD, que puede expresarse de la siguiente manera:

HD (Cadena 1,   Cadena 2) = 4

Ya que hay cuatro bits que difieren en una cadena respecto de la otra.

La Distancia Hamming también se puede expresar como una relación o en porcentaje. En ese caso el resultado se divide por la longitud de la cadena y, si es necesario, se multiplica por 100%

HD (Cadena 1, Cadena 2) = 0,5 o 50%

En el caso de bits distribuidos uniformemente, la Distancia Hamming media de dos cadenas binarias es 50%. La Distancia Hamming de tales cadenas se distribuye binomialmente. La distribución de Distancias Hamming a menudo se muestra en un gráfico donde el eje *x* muestra la probabilidad de ocurrencia y el eje *y* la frecuencia.

6.2 Distribución Binomial

Si los elementos de las cadenas de bits son variables aleatorias, las Distancias Hamming entre diferentes cadenas dan como resultado una distribución binomial. La distribución binomial se define como:

$$\binom{n}{p} p^k (1-p)^{n-k}$$

donde *n* es el número de bits en la cadena de bits, *k* el número de bits diferentes entre dos claves (Distancia Hamming), y *p* la probabilidad de que ocurra que dos bits que ocupan la misma posición sean diferentes.

Es importante destacar lo siguiente: en realidad, para que haya una distribución binomial, la probabilidad de que cualesquiera dos bits sean diferentes tiene que ser constante a lo largo de la clave. Esto sin embargo no es así, y unos bits tienen mayor probabilidad de cambiar que otros. A pesar de ello, es habitual realizar la aproximación binomial y utilizar el valor 'p' como medida de calidad de la PUF.

Para realizar los ajustes de los datos medidos a una distribución binomial se procede de la siguiente forma:

1. Se obtiene la media de la distribución, μ:

   $$\mu = x_1 \cdot P_1 + x_2 \cdot P_2 + \ldots + x_n \cdot P_n$$

   donde $x_1 \ldots x_n$ son los valores de las posibles Distancias Hamming (desde 0 hasta el número de dígitos de las claves) y $P_1 \ldots P_n$ se obtiene como el número de veces que se repite cada Distancia Hamming dividido entre el número total de Distancias Hamming.

2. Una vez se tiene el valor de µ, se obtiene p:

   $$\mu = np \rightarrow p = \frac{\mu}{n}$$

   con *n* el número de dígitos que tiene cada clave (ya que este número coincide con la mayor Distancia Hamming posible, cuando no coincide ningún dígito entre dos claves).

3. Con el valor de *p* y el de *n* ya se puede realizar el ajuste de la distribución binomial.

6.3 Intra-distancia e inter-distancia

Intra-distancia: La intra-distancia, o Distancia Hamming intra-chip, de la respuesta de una PUF es una variable aleatoria que describe la distancia entre dos respuestas de la misma PUF usando el mismo desafío.

Inter-distancia: La inter-distancia, o Distancia Hamming inter-chip, de la respuesta de una PUF es una variable aleatoria que describe la distancia entre dos respuestas de diferentes PUFs usando el mismo desafío. La PUF es siempre la misma, pero se implementa en distintas FPGA.

7. Identificación y autenticación de una entidad basada en PUF

Debido a su combinación de singularidad y reproducibilidad, una PUF en una entidad sirve como una característica de identificación de esa entidad [4]. Además, la imposibilidad física de clonación exhibida por una PUF proporciona garantías de seguridad fuertes, lo que puede utilizarse para fines de autenticación. Sin embargo, para que tenga valor práctico, la seguridad y la solidez de una identificación o autenticación basada en PUF debe cuantificarse en función de sus características de comportamiento verificadas experimentalmente.

- Autenticación de una entidad
  En seguridad de la información, el término autenticación tiene un significado muy amplio. En primer lugar, la autenticación puede relacionarse con entidades o datos. En el primer caso, se habla de autenticación de entidad, mientras que el segundo se llama autenticación de mensaje. Dado que una PUF proporciona una medida de una característica física específica de la entidad, se va a considerar particularmente la autenticación de la entidad.
- Identificación
  Aunque en algunas ocasiones la identificación y la autenticación de una entidad se tratan como sinónimos, aquí se va a hacer una distinción ya que la identificación es un concepto relacionado, pero significativamente más débil que la autenticación. La identificación es el reclamo o declaración de la identidad, sin presentar necesariamente ninguna prueba convincente de ello. Si bien no es estrictamente una técnica de seguridad, ya que no cumple ningún objetivo de seguridad significativo, la identificación tiene cualidades muy útiles:
    - En muchos casos es una condición previa necesaria para la autenticación de la entidad y, por lo tanto, una parte inherente de la mayoría de las técnicas de autenticación de la entidad.
    - Para aplicaciones sin objetivos de seguridad estrictos, la identificación puede ser una condición suficiente, por ejemplo, para aplicaciones que implican el seguimiento de productos en un sistema cerrado.
    - En ciertas situaciones, la identificación es suficiente para lograr la autenticación de la entidad, ya que las condiciones de autenticación se cumplen implícitamente.

En base a estos términos, lo que se busca es lograr la identificación y autenticación de la entidad en función de la singularidad e imprevisibilidad del comportamiento desafío-respuesta de una PUF, e introducir una metodología para cuantificar el rendimiento resultante de identificación y autenticación en términos de seguridad y solidez.

Una característica de identificación inherente es una característica específica de la entidad que surge en el proceso de creación de la entidad. La inherencia de su comportamiento específico

es una de las condiciones clave para que una construcción se considere PUF. Por otro lado, una entidad también puede tener identidades asignadas. Volviendo a la comparación de las PUFs con las huellas dactilares, estas serían inherentes, mientras que el nombre de una persona se asigna después de su nacimiento. Un objeto inanimado también puede tener una identidad asignada, como puede ser un número de serie o un código de barras. En la mayoría de las aplicaciones que requieren identificación de entidad, las identidades asignadas son actualmente una práctica estándar. Por ejemplo, para los chips de silicio digital, las cadenas de bits únicas que se programan en una memoria no volátil incrustada en el chip eran hasta hace poco la única forma de identificar un chip específico. Con la introducción de las PUFs de silicio, es posible usar características únicas inherentes de un chip de silicio, como la identificación.

Las técnicas de identificación basadas en identidades asignadas e inherentes suelen funcionar en dos fases. La primera fase es diferente para ambos tipos:

- Para las identidades asignadas, la primera fase de cualquier técnica de identificación consiste en proporcionar a cada entidad que necesita ser identificada una identidad única permanente. Esta fase se llama de aprovisionamiento.
- Para las identidades inherentes, la primera fase de cualquier técnica de identificación consiste en recopilar las identidades inherentes de cada entidad que necesita ser identificada. Esta fase se llama de inscripción.

La segunda fase es muy similar para ambos tipos y consiste en una entidad que presenta su identidad, ya sea asignada o inherente, cuando se solicita. Esto se llama fase de identificación.

Junto a los conceptos de autenticación e identificación, es conveniente conocer la llamada *fuzzy identification,* que podría traducirse como identificación difusa ya que no hay una seguridad completa de identificar bien una entidad. Esto se ve en la respuesta de una PUF. La falta de claridad de la respuesta de una PUF se muestra más claramente por sus distribuciones de intra e inter-distancia. Cuando se consideran los vectores de respuesta binarios y la Distancia Hamming porcentual como una métrica de distancia, las respuestas aleatorias perfectamente uniformes tendrían una inter-distancia esperada de exactamente el 50%. Resultados experimentales indican que ninguna de las PUFs intrínsecas existentes cumple esta condición, aunque algunas tienen una inter-distancia promedio muy cercana al 50%. De manera equivalente, ninguna PUF intrínseca exhibe una reproducibilidad perfecta con una intra-distancia fija del 0%, y algunas solo son reproducibles hasta una intra-distancia promedio de 10% o incluso más.

Para evaluar la medida en que una respuesta de una PUF se puede utilizar como un identificador inherente, hay que tener en cuenta su falta de definición. Aquí es donde entra en juego la propiedad de identificabilidad de una PUF.

Una PUF exhibe cierto nivel de identificabilidad cuando la intra-distancia esperada es notablemente menor que la inter-distancia esperada. Sin embargo, si hay un solapamiento significativo entre las curvas de ambas distribuciones como se muestra en la figura 7, hay un problema de identificación por la falta de claridad de las respuestas. Cuando una distancia observada entre las respuestas de la fase de inscripción y la fase de identificación cae en esta región superpuesta, puede corresponder a medidas de una misma instancia PUF o de distintas instancias PUF, y no hay forma de distinguir entre ambos casos.

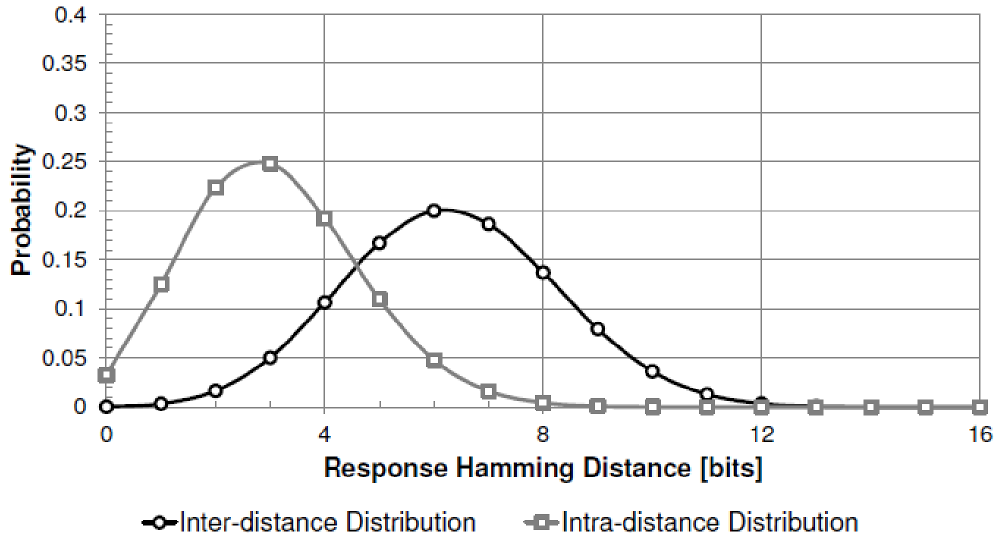

Figura 7: Ejemplo de solapamiento entre las curvas de intra-distancia e inter-distancia

En un sistema de identificación práctico para identidades con falta de claridad, es necesario determinar un umbral de distancia de respuesta bastante pragmático. Se supone que las distancias menores o iguales a este umbral corresponden a medidas de una misma instancia PUF, mientras que las distancias por encima de este umbral corresponden a medidas de distintas instancias PUF. Este umbral se llama el umbral de identificación.

Durante la fase de identificación de un sistema de identificación basado en PUF, la respuesta generada por una entidad se compara con una lista de respuestas inscritas. Cuando se encuentra una respuesta inscrita cuya distancia a la respuesta presentada es menor o igual al umbral de identificación, la entidad se identifica como la entrada coincidente en la lista. Está claro que un sistema de identificación difuso basado en un umbral de identificación tan pragmático no es absolutamente confiable, especialmente cuando hay una gran superposición entre las distribuciones de inter e intra-distancia. Al comparar una respuesta con una respuesta de la lista de inscripción, pueden surgir cuatro situaciones posibles:

1. La entidad presentada es la misma entidad que produjo la respuesta inscrita y logra reproducir la respuesta inscrita con una intra-distancia menor que el umbral de identificación. La entidad presentada está correctamente identificada. Esto se llama verdadera aceptancia.

2. La entidad presentada es la misma entidad que produjo la respuesta inscrita pero no puede reproducir la respuesta inscrita con una intra-distancia menor que el umbral de identificación. La entidad presentada es rechazada por error. Esto se llama falso rechazo.

3. La entidad presentada no es la misma entidad que produjo la respuesta inscrita, pero se produce una respuesta cuya inter-distancia a la respuesta inscrita es menor que el umbral de identificación. La entidad presentada se identifica por error. Esto se llama falsa aceptancia.

4. La entidad presentada no es la misma entidad que produjo la respuesta inscrita, y produce una respuesta cuya inter-distancia es mayor que el umbral de identificación. La entidad presentada se rechaza correctamente. Esto se llama verdadero rechazo.

Las falsas aceptancias y los falsos rechazos son indeseables para un sistema de identificación práctico. La probabilidad de que un intento de identificación aleatoria resulte en uno de estos casos se expresa respectivamente como la tasa de falsa aceptancia (FAR) y como la tasa de falso rechazo (FRR) del sistema. FAR expresa la seguridad de un sistema de identificación, ya que un FAR bajo significa que hay poco riesgo de identificación errónea que podría conducir a problemas de seguridad. FRR por otro lado expresa la robustez o usabilidad de un sistema, ya que expresa el riesgo de rechazar injustamente entidades legítimas, lo que sería muy poco práctico. Para un sistema de identificación utilizable, tanto FAR como FRR deben ser tan pequeños como sea posible, pero no se pueden minimizar al mismo tiempo. Esta minimización suele depender de la aplicación, pero se suele adoptar el compromiso de que tengan el mismo valor. Debe realizarse un intercambio aceptable entre seguridad y usabilidad.

Para un sistema de identificación dado, FAR y FRR dependen de la elección del valor del umbral de identificación que se denomina t_{id}. Un umbral alto minimiza el riesgo de un falso rechazo, pero aumenta la probabilidad de falsas aceptancias, y viceversa para un umbral bajo. Cuando se conocen las distribuciones de las inter e intra-distancias de las PUFs consideradas, las relaciones respectivas entre FAR, FRR y t_{id} se pueden calcular:

- FAR es la probabilidad de que la inter-distancia sea menor o igual a t_{id}. Esto es equivalente a la evaluación de la función de distribución acumulativa de la inter-distancia en t_{id}.
- FRR es la probabilidad de que la intra-distancia sea mayor que t_{id}. Esto es equivalente al complemento de la evaluación de la función de distribución acumulativa de la intra-distancia en t_{id}.

8. Caracterización experimental

8.1 Medida de la intra-distancia de una GARO-PUF en una FPGA para distintas temperaturas

Para estudiar la calidad de la PUF de oscilador de anillo de Galois, se implementa usando Vivado en la FPGA PYNQ-Z2 el circuito que se ha explicado previamente, constituido por 100 osciladores de 5 LUTs cada uno. Se toman medidas desde una temperatura de -20°C hasta 80°C, cada 10°C, lo que supone un total de 11 medidas. Para cada una de estas 11 medidas, se generan 20 claves de 99 dígitos cada una, esto es, la comparación entre osciladores se realiza veinte veces, y el número de dígitos es el número de osciladores menos uno.

Una vez están todas las medidas hechas, se tiene un fichero de texto correspondiente a cada temperatura, y en cada uno de ellos hay 20 claves binarias. A continuación, se pasa a realizar el estudio de estos datos. En primer lugar, se obtiene la intra-distancia, esto es, la Distancia Hamming entre las claves que ha generado la PUF. Al tener 20 claves, se obtienen 190 valores de intra-distancia (combinaciones de 20 claves tomadas de dos en dos sin repetición), de los cuales se calcula el valor medio con su error y la varianza. La varianza de una variable aleatoria es una medida de dispersión definida como la esperanza del cuadrado de la desviación de dicha variable respecto a su media. El error de la media se obtiene dividiendo la desviación típica (raíz

cuadrada de la varianza) entre la raíz cuadrada del número de medidas. Las expresiones correspondientes a la media y la varianza son las siguientes:

$$\text{Media muestral: } \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$\text{Varianza muestral: } s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

Los resultados obtenidos se muestran en la Tabla 1 y se representan en las Figuras 8 y 9.

TEMPERATURA (°C)	MEDIA DISTANCIAS HAMMING	VARIANZA MUESTRAL
-20	4,49 ± 0,17	5,40
-10	2,66 ± 0,09	1,72
0	6,78 ± 0,14	3,89
10	4,89 ± 0,14	3,98
20	6,01 ± 0,14	4,00
30	5,48 ± 0,15	4,21
40	4,59 ± 0,12	2,78
50	5,33 ± 0,12	2,80
60	4,10 ± 0,12	2,78
70	3,11 ± 0,10	1,89
80	3,39 ± 0,10	1,97

Tabla 1: Medidas en la GARO-PUF con 20 claves

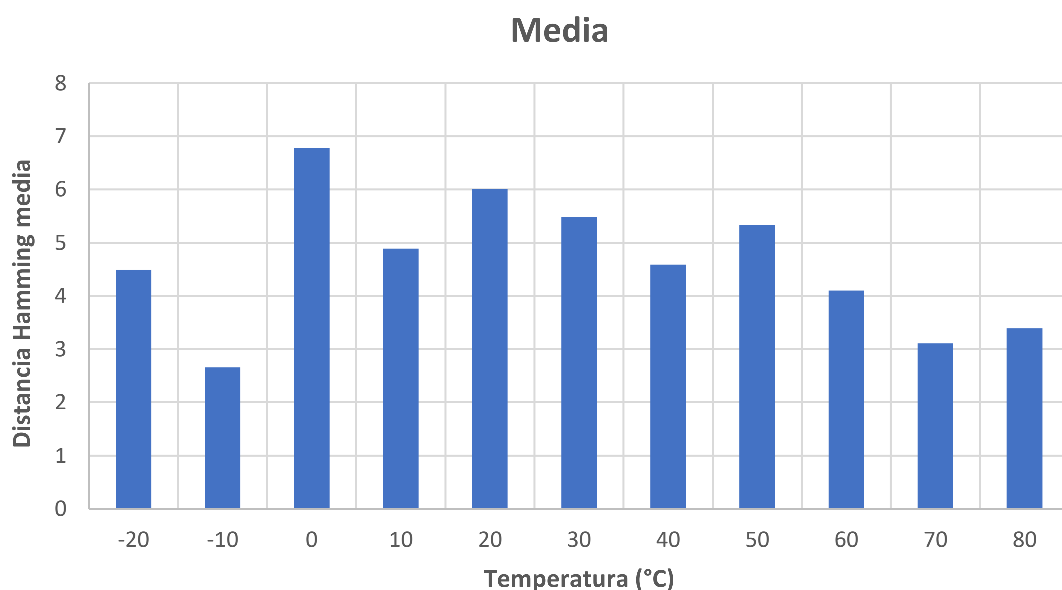

Figura 8: Valor medio de la Distancia Hamming en la GARO-PUF para diferentes temperaturas y 20 claves

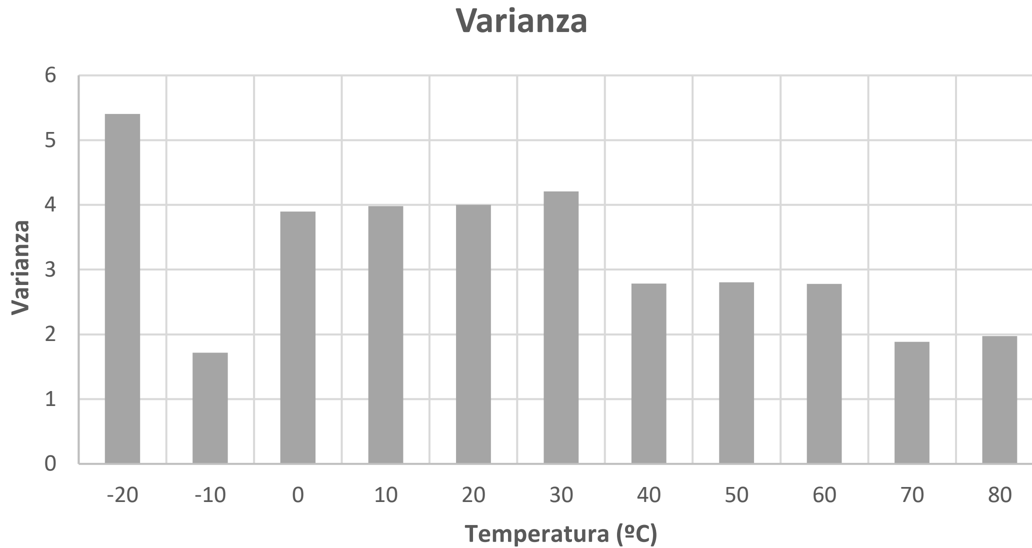

Figura 9: Varianza de las Distancias Hamming en la GARO-PUF para diferentes temperaturas y 20 claves

Teniendo en cuenta que el valor de la intra-distancia puede variar entre 0 y 99, y que, de los valores medios obtenidos, el menor es 2,66 y el mayor 6,78, sin apreciarse una tendencia directamente relacionada con la variación de temperatura, en primera instancia se puede suponer que la PUF no pierde robustez en el rango de temperaturas estudiado.

8.2 Medida de la intra-distancia de una RO-PUF en una FPGA para distintas temperaturas

Se quiere comparar la robustez de la PUF de oscilador de anillo de Galois con una PUF de oscilador de anillo simple. Para ello se lleva a cabo el mismo estudio que antes, generando 20 claves de 99 dígitos para las 11 mismas temperaturas, pero lo que se implementa en la FPGA son osciladores de anillo simples con 5 inversores cada uno.

Los resultados obtenidos se muestran en la Tabla 2 y se representan en las Figuras 10 y 11.

TEMPERATURA (°C)	MEDIA DISTANCIAS HAMMING	VARIANZA MUESTRAL
-20	1,00 ± 0,11	2,39
-10	0,60 ± 0,05	0,40
0	1,10 ± 0,06	0,66
10	1,20 ± 0,05	0,46
20	0,10 ± 0,02	0,09
30	1,67 ± 0,09	1,49
40	0,98 ± 0,07	0,82
50	1,30 ± 0,08	1,09
60	2,49 ± 0,08	1,31
70	1,12 ± 0,05	0,55
80	1,32 ± 0,06	0,74

Tabla 2: Medidas en la RO-PUF con 20 claves

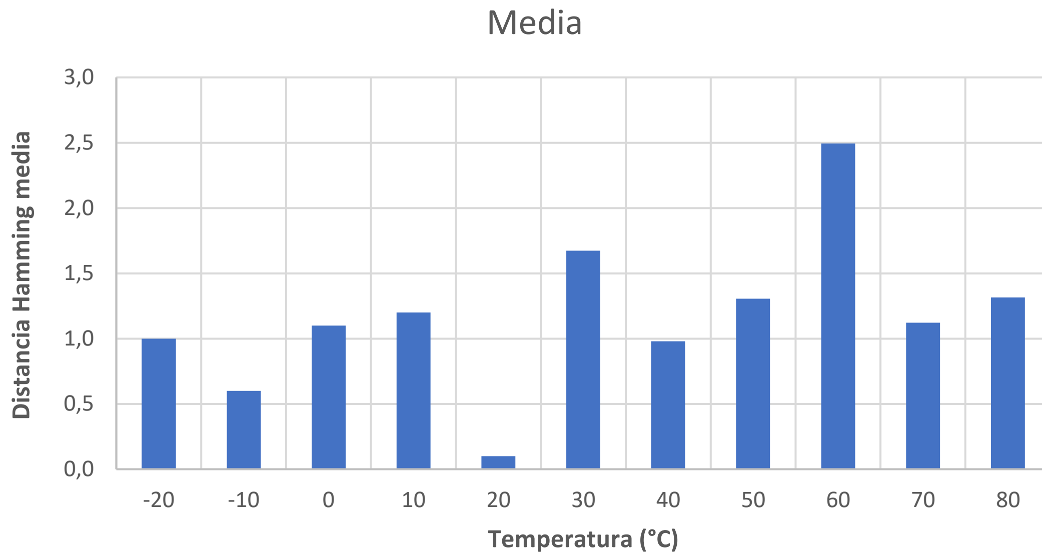

Figura 10: Valor medio de la Distancia Hamming en la RO-PUF para diferentes temperaturas y 20 claves

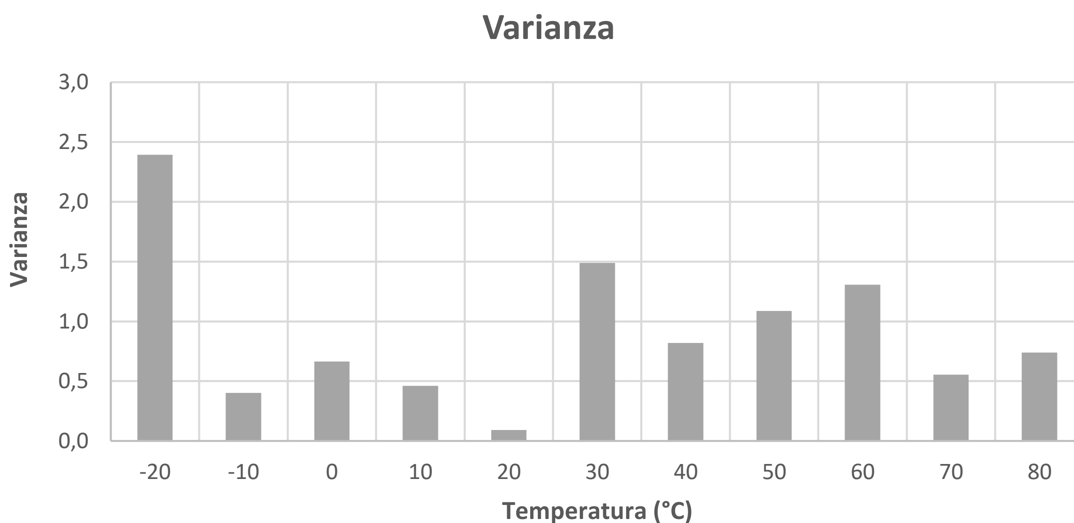

Figura 11: Varianza de las Distancias Hamming en la RO-PUF para diferentes temperaturas y 20 claves

A la vista de los resultados, en los cuales no se aprecia un gran efecto de la temperatura, se decide volver a realizar las medidas, midiendo ahora un mayor número de claves, 100, en lugar de 20 como en los casos anteriores, disminuyendo el tiempo de retardo de 1 segundo a 1 milisegundo. El tiempo de retardo viene dado por la expresión t = TN, donde T es el período natural del oscilador y N un número de períodos. Dependiendo del tiempo de retardo que se elija, se medirá el correspondiente número de períodos, de tal forma que cuanto mayor sea el tiempo de retardo, mayor será el número de períodos medidos y mayor será la precisión de la medida. En este caso al disminuir el tiempo de retardo se pierde precisión en cada una de las medidas y se decide a cambio aumentar el número de medidas.

Los resultados obtenidos se muestran en la Tabla 3 y se representan en las Figuras 12 y 13.

TEMPERATURA (°C)	MEDIA DISTANCIAS HAMMING	VARIANZA MUESTRAL
-20	0,791 ± 0,009	0,462
-10	0,059 ± 0,003	0,056
0	0,080 ± 0,003	0,076
10	0,040 ± 0,003	0,038
20	0,237 ± 0,006	0,191
30	0,134 ± 0,004	0,119
40	0,536 ± 0,008	0,362
50	0,854 ± 0,010	0,545
60	1,081 ± 0,011	0,697
70	0,863 ± 0,010	0,544
80	0,767 ± 0,011	0,584

Tabla 3: Medidas en la RO-PUF con 100 claves

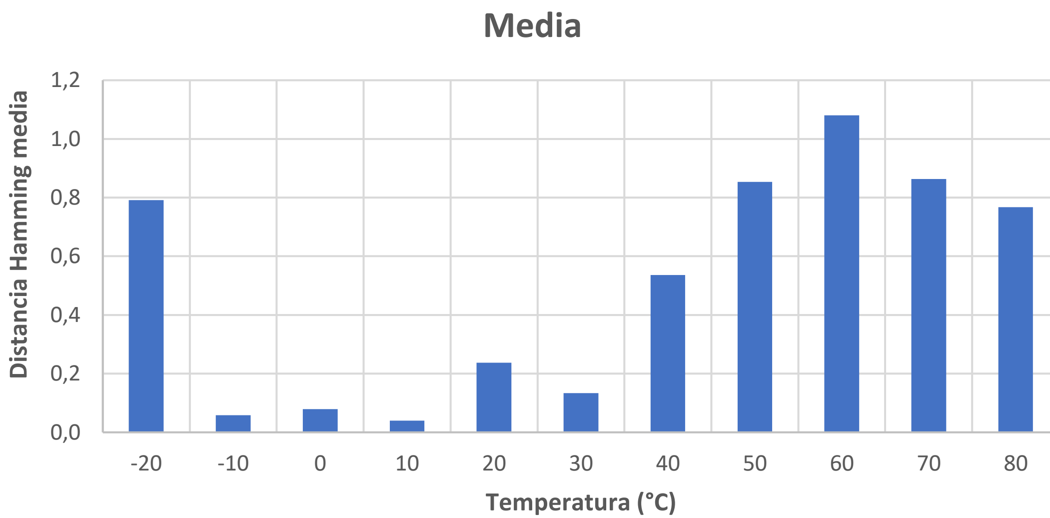

Figura 12: Valor medio de la Distancia Hamming en la RO-PUF para diferentes temperaturas y 100 claves

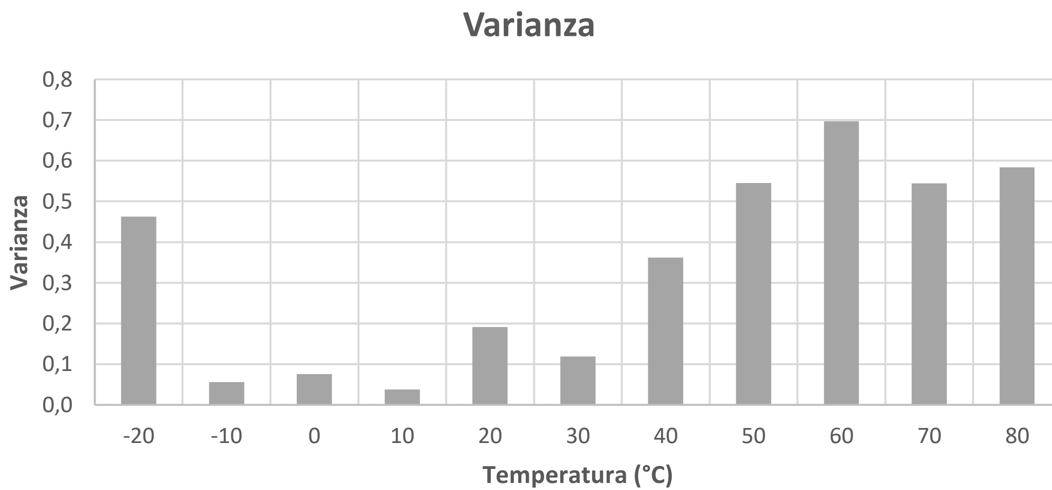

Figura 13: Varianza de las Distancias Hamming en la RO-PUF para diferentes temperaturas y 100 claves

Se puede apreciar tanto en la representación de la media como de la varianza, que desde -10°C hasta 30°C la medida se mantiene aproximadamente constante con la variación de la temperatura, y a partir de los 30°C y hasta los 60°C, se observa una tendencia de crecimiento al aumentar la temperatura. Sin embargo, el valor a -20 °C es mayor que los posteriores valores, y al pasar de 60°C a 70°C y 80°C disminuye. De esta forma, no se aprecia realmente una tendencia clara de la variación de la Distancia Hamming con la temperatura.

Si se comparan los resultados obtenidos con los dos tipos de PUF, se observa que no hay una gran diferencia, siendo la variación entre la Distancia Hamming media máxima y mínima ligeramente mayor en la GARO-PUF, ya que se produce una variación de 4,12 unidades frente a 2,39 unidades en la RO-PUF midiendo 20 claves y 1,04 unidades midiendo 100 claves. Estos valores no son destacables ni muy distintos debido a que las claves son de 99 dígitos.

8.3 Medida de la inter-distancia

Una vez realizado el estudio de la intra-distancia de la PUF en una misma FPGA, se estudia la inter-distancia, por lo que se pasa de usar una única FPGA a usar 20 placas y comparar las claves generadas entre sí. Para cada placa se implementa el oscilador de anillo de Galois de 5 LUT y se generan 20 claves. Hasta ahora se ha procedido de la misma manera que en la medida de las intra-distancias. La diferencia es que ahora, con las 20 claves generadas, se obtiene la que sería la clave más probable de cada PUF. Para ello, tras contar el número de ceros y unos que hay en cada una de las posiciones de las claves, si hay más unos, a esa posición se le asigna un 1, y si hay más ceros, se le asigna un 0. En el caso en el que haya el mismo número de ceros y unos se le asigna una x por convenio. Esto se hace para no tener preferencia entre 0 o 1, así, a la hora de medir la inter-distancia, este dígito será distinto siempre de su correspondiente en otra clave. Hecho esto, se tienen 20 claves, cada una correspondiente a una FPGA. Finalmente se realiza el estudio de estas 20 claves calculando la Distancia Hamming entre ellas.

8.4 Identificación y autenticación

Una vez se tienen todos los resultados de intra-distancias de las claves generadas por la PUF en cada una de las placas y de las inter-distancias entre las claves generadas por la PUF implementada en cada una de las placas se pasa a representar las curvas correspondientes a intra-distancia e inter-distancia para cada una de las placas. Se representa mediante un histograma la frecuencia de las intra-distancias, esto es, cuántas veces se repite cada una. En el eje horizontal se muestra la Distancia Hamming y en el eje vertical el número de veces que se repite cada una de ellas. La distribución de Distancias Hamming de una PUF puede ajustarse a una distribución binomial.

La Figura 14 muestra la representación para la placa 1, tanto de las medidas obtenidas como del ajuste a la distribución binomial.

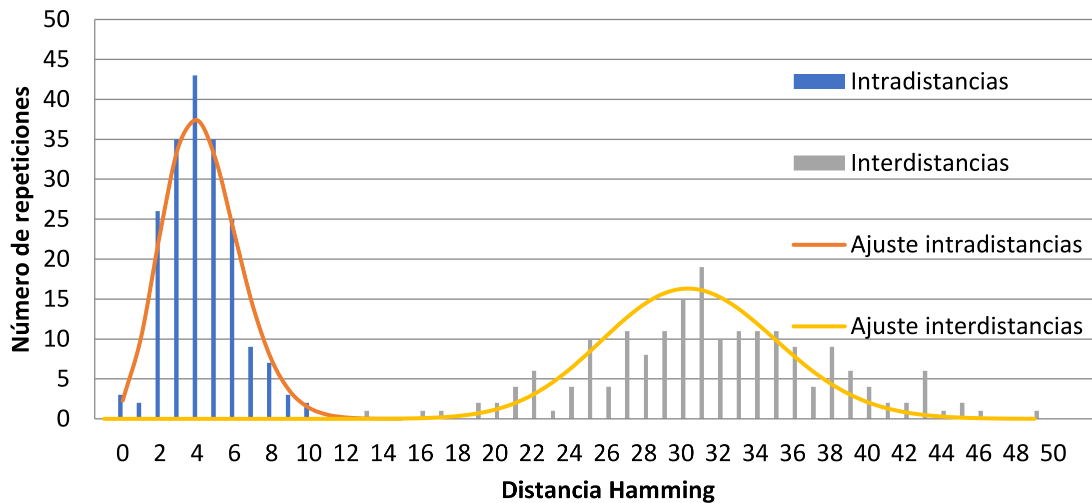

Figura 14: Ajuste a una distribución binomial de las intra-distancias y las inter-distancias para la placa 1

El resultado ha sido similar en las 20 placas. Basándose en el estudio hecho en el apartado sobre identificación y autenticación se puede realizar el siguiente análisis: En primer lugar, el valor medio de las inter-distancias no está en el 50%. La inter-distancia máxima sería 99, al estar cada clave compuesta por 99 bits, y en este la inter-distancia media está en torno a 30, es decir, en torno a un 30%. En cuanto a las intra-distancias, pasa algo similar a lo esperado, el promedio no es de 0%, ya que esto sería una reproducibilidad perfecta, lo cual es imposible, y el valor obtenido está en torno a un 5%.

Dicho esto, la intra-distancia esperada es notablemente menor que la inter-distancia esperada, lo cual permite que las curvas no se solapen y no haya un posible problema de identificación por falta de claridad de las respuestas, exhibiendo así la PUF una buena identificabilidad.

9. Conclusión

Este Trabajo Fin de Grado se ha centrado en el estudio de las funciones físicamente no clonables (PUFs) y más específicamente en un caso particular de estas: las PUFs de oscilador de anillo de Galois, aunque en la parte experimental también se trabajó con los osciladores de anillo simples.

En cuanto a las conclusiones que se pueden extraer del trabajo destacan las siguientes:

- No se puede llegar a afirmar una posible o no influencia de la temperatura ya que no se aprecia una tendencia clara de variación de las medidas al variar la temperatura.
- No hay una gran diferencia entre las medidas obtenidas en la GARO-PUF y la RO-PUF.
- En cuanto a la parte de identificabilidad, los resultados obtenidos se pueden considerar positivos, debido a su semejanza con lo esperado.

El estudio de las PUFs de oscilador de anillo de Galois podría continuar en un futuro. Entre las líneas que se podrían seguir cabe citar un estudio como el de este trabajo en el que se realizan medidas con variaciones de temperatura, pero empleando osciladores de anillos con más LUTs o implementando la PUF en diferentes posiciones de la FPGA. Otra posibilidad sería realizar las mediciones variando el voltaje de alimentación de la placa en lugar de variando la temperatura.

10. Bibliografía

[1] Wang, X., & Tehranipoor, M. (2010). Novel physical unclonable function with process and environmental variations*. Design, Automation & Test in Europe Conference & Exhibition*, 1065-1070.

[2] Böhm, C., & Hofer, M. (2013). *Physical unclonable functions in theory and practice.* Nueva York: Springer-Verlag.

[3] Handschuch, H., Schrijen, G-J., & Tuyls, T. (2011). Hardware intrinsec security from physically unclonable functions. En *Towards Hardware-Intrinsic Security* (págs. 39-53). Berlín, Heidelberg: Springer

[4] Maes, R. (2012). *Physically unclonable functions: constructions, properties and applications.* Leuven: Arenberg Doctoral School of Science, Enginering &Technology.

[5] Golic, J. D. (2006). New methods for digital generation and postprocessing of random data. *IEEE transactions on computers, 55*(10), 1217-1229.

[6] García Bosque, M., Díez Señorans, G., Sánchez Azqueta, C., Celma Pueyo, S. (2020, enviado). Proposal and Analysis of a Novel Class of PUFs Based on Galois Ring Oscillators. *IEEE Transactions on Circuits and Systems I.*

[7] Xilinx (2016). *7 Series FPGAs Configurable Logic Block User Guide*

[8] Maxfield, C. (2004). The *design warrior's guide to FPGAs: devices, tools and flows.* Elsevier

[9] Zynq (2018). *PYNQ-Z2 Reference Manual v1.0*

Universidad Zaragoza

Trabajo Fin de Grado: Anexos

Generador hardware de claves basado en funciones físicamente no clonables

Autor

Daniel Gil Marco

Directores

Carlos Sánchez Azqueta

Guillermo Díez Señorans

Facultad de Ciencias

Departamento de Ingeniería Electrónica y Comunicaciones

Junio 2020

ANEXO A: Códigos VHDL para el diseño de la PUF en Vivado

A.1. Módulo Clock Divider

```
entity Clock_Divider is
            Port
            (master_clk : in STD_LOGIC;
            Clk_100kHz : out STD_LOGIC);
end Clock_Divider;

architecture Behavioral of Clock_Divider is
signal output: STD_LOGIC:='0';
begin
    process(master_clk)
    variable counter_100kHz: integer range 0 to (499):=0;

    begin
         if (rising_edge(master_clk)) then
            if (counter_100kHz=499) then
            counter_100kHz:=0;
            output<=not output;
            else
            counter_100kHz:=counter_100kHz+1;
            end if;
        end if;
    end process;
    Clk_100kHz<=output;
end Behavioral;
```

A.2. Módulo de selección de osciladores

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity OscillatorSelector is
    port (
        selector: in integer range 0 to 98;
        array_of_bits : in STD_LOGIC_VECTOR (99 downto 0);
        output_bit_1: out STD_LOGIC;
        output_bit_2: out STD_LOGIC
    );
end OscillatorSelector;

architecture Behavioral of OscillatorSelector is

begin

  with selector select
            output_bit_1 <=
                    array_of_bits(0) when 0,
                    array_of_bits(1) when 1,
                    array_of_bits(2) when 2,
                    array_of_bits(3) when 3,
                    array_of_bits(4) when 4,
                    array_of_bits(5) when 5,
                    array_of_bits(6) when 6,
                    array_of_bits(7) when 7,
                    array_of_bits(8) when 8,
                    array_of_bits(9) when 9,
                    array_of_bits(10) when 10,
                    array_of_bits(11) when 11,
                    array_of_bits(12) when 12,
                    array_of_bits(13) when 13,
                    array_of_bits(14) when 14,
                    array_of_bits(15) when 15,
                    array_of_bits(16) when 16,
                    array_of_bits(17) when 17,
                    array_of_bits(18) when 18,
                    array_of_bits(19) when 19,
                    array_of_bits(20) when 20,
                    array_of_bits(21) when 21,
                    array_of_bits(22) when 22,
                    array_of_bits(23) when 23,
                    array_of_bits(24) when 24,
                    array_of_bits(25) when 25,
                    array_of_bits(26) when 26,
                    array_of_bits(27) when 27,
                    array_of_bits(28) when 28,
                    array_of_bits(29) when 29,
                    array_of_bits(30) when 30,
                    array_of_bits(31) when 31,
                    array_of_bits(32) when 32,
```

```
array_of_bits(33) when 33,
array_of_bits(34) when 34,
array_of_bits(35) when 35,
array_of_bits(36) when 36,
array_of_bits(37) when 37,
array_of_bits(38) when 38,
array_of_bits(39) when 39,
array_of_bits(40) when 40,
array_of_bits(41) when 41,
array_of_bits(42) when 42,
array_of_bits(43) when 43,
array_of_bits(44) when 44,
array_of_bits(45) when 45,
array_of_bits(46) when 46,
array_of_bits(47) when 47,
array_of_bits(48) when 48,
array_of_bits(49) when 49,
array_of_bits(50) when 50,
array_of_bits(51) when 51,
array_of_bits(52) when 52,
array_of_bits(53) when 53,
array_of_bits(54) when 54,
array_of_bits(55) when 55,
array_of_bits(56) when 56,
array_of_bits(57) when 57,
array_of_bits(58) when 58,
array_of_bits(59) when 59,
array_of_bits(60) when 60,
array_of_bits(61) when 61,
array_of_bits(62) when 62,
array_of_bits(63) when 63,
array_of_bits(64) when 64,
array_of_bits(65) when 65,
array_of_bits(66) when 66,
array_of_bits(67) when 67,
array_of_bits(68) when 68,
array_of_bits(69) when 69,
array_of_bits(70) when 70,
array_of_bits(71) when 71,
array_of_bits(72) when 72,
array_of_bits(73) when 73,
array_of_bits(74) when 74,
array_of_bits(75) when 75,
array_of_bits(76) when 76,
array_of_bits(77) when 77,
array_of_bits(78) when 78,
array_of_bits(79) when 79,
array_of_bits(80) when 80,
array_of_bits(81) when 81,
array_of_bits(82) when 82,
array_of_bits(83) when 83,
array_of_bits(84) when 84,
array_of_bits(85) when 85,
```

```
                array_of_bits(86) when 86,
                array_of_bits(87) when 87,
                array_of_bits(88) when 88,
                array_of_bits(89) when 89,
                array_of_bits(90) when 90,
                array_of_bits(91) when 91,
                array_of_bits(92) when 92,
                array_of_bits(93) when 93,
                array_of_bits(94) when 94,
                array_of_bits(95) when 95,
                array_of_bits(96) when 96,
                array_of_bits(97) when 97,
                array_of_bits(98) when 98,

                '0' when others;
with selector select
            output_bit_2 <=
                array_of_bits(1) when 0,
                array_of_bits(2) when 1,
                array_of_bits(3) when 2,
                array_of_bits(4) when 3,
                array_of_bits(5) when 4,
                array_of_bits(6) when 5,
                array_of_bits(7) when 6,
                array_of_bits(8) when 7,
                array_of_bits(9) when 8,
                array_of_bits(10) when 9,
                array_of_bits(11) when 10,
                array_of_bits(12) when 11,
                array_of_bits(13) when 12,
                array_of_bits(14) when 13,
                array_of_bits(15) when 14,
                array_of_bits(16) when 15,
                array_of_bits(17) when 16,
                array_of_bits(18) when 17,
                array_of_bits(19) when 18,
                array_of_bits(20) when 19,
                array_of_bits(21) when 20,
                array_of_bits(22) when 21,
                array_of_bits(23) when 22,
                array_of_bits(24) when 23,
                array_of_bits(25) when 24,
                array_of_bits(26) when 25,
                array_of_bits(27) when 26,
                array_of_bits(28) when 27,
                array_of_bits(29) when 28,
                array_of_bits(30) when 29,
                array_of_bits(31) when 30,
                array_of_bits(32) when 31,
                array_of_bits(33) when 32,
                array_of_bits(34) when 33,
                array_of_bits(35) when 34,
                array_of_bits(36) when 35,
                array_of_bits(37) when 36,
                array_of_bits(38) when 37,
```

```
array_of_bits(39) when 38,
array_of_bits(40) when 39,
array_of_bits(41) when 40,
array_of_bits(42) when 41,
array_of_bits(43) when 42,
array_of_bits(44) when 43,
array_of_bits(45) when 44,
array_of_bits(46) when 45,
array_of_bits(47) when 46,
array_of_bits(48) when 47,
array_of_bits(49) when 48,
array_of_bits(50) when 49,
array_of_bits(51) when 50,
array_of_bits(52) when 51,
array_of_bits(53) when 52,
array_of_bits(54) when 53,
array_of_bits(55) when 54,
array_of_bits(56) when 55,
array_of_bits(57) when 56,
array_of_bits(58) when 57,
array_of_bits(59) when 58,
array_of_bits(60) when 59,
array_of_bits(61) when 60,
array_of_bits(62) when 61,
array_of_bits(63) when 62,
array_of_bits(64) when 63,
array_of_bits(65) when 64,
array_of_bits(66) when 65,
array_of_bits(67) when 66,
array_of_bits(68) when 67,
array_of_bits(69) when 68,
array_of_bits(70) when 69,
array_of_bits(71) when 70,
array_of_bits(72) when 71,
array_of_bits(73) when 72,
array_of_bits(74) when 73,
array_of_bits(75) when 74,
array_of_bits(76) when 75,
array_of_bits(77) when 76,
array_of_bits(78) when 77,
array_of_bits(79) when 78,
array_of_bits(80) when 79,
array_of_bits(81) when 80,
array_of_bits(82) when 81,
array_of_bits(83) when 82,
array_of_bits(84) when 83,
array_of_bits(85) when 84,
array_of_bits(86) when 85,
array_of_bits(87) when 86,
array_of_bits(88) when 87,
array_of_bits(89) when 88,
array_of_bits(90) when 89,
array_of_bits(91) when 90,
```

```
                    array_of_bits(92) when 91,
                    array_of_bits(93) when 92,
                    array_of_bits(94) when 93,
                    array_of_bits(95) when 94,
                    array_of_bits(96) when 95,
                    array_of_bits(97) when 96,
                    array_of_bits(98) when 97,
                    array_of_bits(99) when 98,
                    '0' when others;

end Behavioral;
```

A.3. Módulo suma

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Suma is
    port (
    sampled_bit: in STD_LOGIC;
    master_clk: in STD_LOGIC;
    Clk_100kHz: in STD_LOGIC;
    reset: in STD_LOGIC;
    sum : out INTEGER range 0 to 1000000:=0

);
end Suma;

architecture Behavioral of Suma is
signal sum_buffer: INTEGER range 0 to 1000000;

begin
sum<=sum_buffer;
    process (master_clk, Clk_100kHz)
        begin
        if reset='1' then
        sum_buffer<=0;
        else
             if (rising_edge(Clk_100kHz)) then
                if (sampled_bit='1') then
                sum_buffer<=sum_buffer+1;
                end if;
             end if;
         end if;
     end process;

end Behavioral;
```

A.4. Módulo comparador

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Comparator is
    port (
    sum1 : in INTEGER range 0 to 1000000:=0;
    sum2 : in INTEGER range 0 to 1000000:=0;
    master_clk: in STD_LOGIC;
    trigger: in STD_LOGIC;
    comp: out STD_LOGIC;
    igual: out STD_LOGIC;
    reset: in STD_LOGIC
    );
end Comparator;

architecture Behavioral of Comparator is

begin

    process (master_clk)
        begin
        if reset='1' then
        comp<='0';
        igual<='0';
        else
             if (rising_edge(trigger)) then
                if (sum1<sum2) then
                comp<='0';
                igual<='0';
                else
                    if (sum1>sum2) then
                    comp<='1';
                    igual<='0';
                        else
                        comp<='1';
                        igual<='1';
                     end if;
                 end if;
              end if;
          end if;
     end process;

end Behavioral;
```

A.5. Módulo Control

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Control is
    port (
        master_clk: in STD_LOGIC;
        arduino: in STD_LOGIC;
        reset: in STD_LOGIC;
        selector: out integer range 0 to 98;
        measure_done: out STD_LOGIC;
        done: out STD_LOGIC
    );
end Control;

architecture Behavioral of Control is
signal selector_buffer: INTEGER range 0 to 98;
signal arduino_registrado: STD_LOGIC;
signal arduino_registrado2: STD_LOGIC;

    type t_state is ( MEASURE, WAIT_ARDUINO, FINISH);
    signal my_state: t_state;
begin
    process(master_clk)
    begin
        if rising_edge(master_clk) then
        arduino_registrado<=arduino;
        arduino_registrado2<=arduino_registrado;
        end if;
    end process;
    process(master_clk)

    begin
        if rising_edge(master_clk) then
         if (reset='1') then
                    done<='0';
                    measure_done<='0';
                    selector_buffer <= 0;
                    my_state<= MEASURE;
         else
            case my_state is
                when MEASURE =>

                        if (arduino_registrado2='1') then
                        my_state<= WAIT_ARDUINO;
                        measure_done<='1';
                        else
                        my_state<= MEASURE;
                        end if;

                when WAIT_ARDUINO =>
                measure_done<='0';
```

```
                        if (arduino_registrado2='1')  then
                        my_state<= WAIT_ARDUINO;
                             if (selector_buffer=98) then
                              my_state<= FINISH;
                              end if;
                        else

                        selector_buffer<=selector_buffer+1;
                        my_state<= MEASURE;

                        end if;

                  when FINISH =>
                  done<='1';

                        my_state<= FINISH;

              end case;

          end if;
          end if;


      end process;

      selector<=selector_buffer;
  end Behavioral;
```

ANEXO B: Códigos en Arduino para la generación de claves

B.1. Oscilador de anillo de Galois

```
#define N_claves 20

#define RETARDO 1000000


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

void setup() {
  //put your setup code here, to run once:
  pinMode(A0, INPUT); //comp
  pinMode(A2, INPUT); //igual
  pinMode(A4, INPUT); //fin

  pinMode(A1, OUTPUT); //rst
  pinMode(A3, OUTPUT); //arduino

  //pinMode(A5, INPUT); //calculando

  Serial.begin(9600);
}

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


  Serial.print(" N_claves = "); Serial.println(N_claves);
  Serial.flush();

  for(i=0;i<N_claves;i++)
  {
    contador = 0;
    digitalWrite(A1,HIGH); delay(10);
    digitalWrite(A1,LOW); //reseteo

    fin = digitalRead(A4);
    while(fin==0)
    {
           if(RETARDO>0)
          {
            if(RETARDO < 100000)
              delayMicroseconds(RETARDO);
            else
              delay(RETARDO/1000);
          }
          digitalWrite(A3,HIGH);

          comp = digitalRead(A0);
          igual = digitalRead(A2);

          if(igual) result = 2;
          else result = comp;

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

```
            delay(100);
            fin = digitalRead(A4);
            digitalWrite(A3,LOW);
      }

        Serial.println();
        digitalWrite(A1,HIGH); delay(10);
        digitalWrite(A1,LOW);

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

B.2. Oscilador de anillo simple

```
#define N_claves 100

#define RETARDO 10000


unsigned int i, j, comp, igual, fin, result, permacc, contador;

void setup() {
  //put your setup code here, to run once:
  pinMode(A0, INPUT); //comp
  pinMode(A1, INPUT); //igual
  pinMode(A2, INPUT); //fin
  pinMode(A3, INPUT); //permacc

  pinMode(A4, OUTPUT); //rst
  pinMode(A5, OUTPUT); //solicitud

  Serial.begin(9600);
}

void loop() {
    // put your main code here, to run repeatedly:
  Serial.print(" N_claves = ");
  Serial.println(N_claves);
  Serial.flush();

  for(i=0;i<N_claves;i++)
  {
    digitalWrite(A4,HIGH);

    fin = 0;
    contador=0;
    while(fin==0)
    {
          digitalWrite(A5,LOW);
          if(contador==0) digitalWrite(A4,LOW);

          if(RETARDO>0)
          {
            if(RETARDO < 100000)
              delayMicroseconds(RETARDO);
            else
              delay(RETARDO/1000);
          }

          digitalWrite(A5,HIGH);
          for(j=0; j<1; j+=0){ if(digitalRead(A3)) break; }

          comp = digitalRead(A0);
          igual = digitalRead(A1);
          fin = digitalRead(A2);

          if(igual) result = 2;
          else result = comp; contador++;
```

```
            Serial.print(result);
      }
      Serial.println();
    }
    Serial.println("- fin -");
    Serial.end();
    for(i=0;i<1;i+=0){}
}
```

ANEXO C: Código en C para el cálculo de Distancias Hamming y estadística

Este primer código lee un fichero con las claves y calcula las Distancias Hamming entre todas ellas, generando un fichero. Leyendo ese fichero, cuenta cuántas veces se repite cada una de las Distancias Hamming, escribiéndolo en otro fichero.

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

#define N_claves 20
#define N_digitos 99
#define combinaciones 190

int main ()
{
    FILE *claves, *salida, *distancias_hamming, *frecuencias_salida;
    char nombre_fichero_claves[100], nombre_fichero_salida[100],
         nombre_fichero_distancias_hamming[100],nombre_fichero_salida_frecuencias[100];
    int matriz[N_claves][N_digitos], i, j, k,l, digito_leido, distancia_hamming_leida,
        contador=0, frecuencias[combinaciones],distancia_hamming,numero_repeticiones=0;


    printf("Introducir nombre fichero claves: ");
    scanf("%s", nombre_fichero_claves);

    claves=fopen(nombre_fichero_claves, "r");

    if (claves==NULL)
    {
        printf("Error al abrir el fichero\n");
        return 0;
    }

    printf("Introducir nombre fichero salida: ");
    scanf("%s", nombre_fichero_salida);

    salida=fopen(nombre_fichero_salida, "w");

    if (salida==NULL)
    {
        printf("Error al abrir el fichero\n");
        return 0;
    }


    for(i=0; i<N_claves; i++)
    {
        for(j=0; j<N_digitos; j++)
            {
            fscanf(claves, "%d", &digito_leido);
            matriz[i][j]=digito_leido;
            }
    }
```

```
 for  (i=0; i<N_claves; i++)
      {
           for(k=1; k<(N_claves-i) ; k++)
                {
                     for(j=0;j<N_digitos;j++)
                          {
                               if (matriz[i][j] != matriz [i+k][j] )
                                    contador=contador+1;
                          }
                          fprintf(salida, "%d\n", contador);
                     contador=0;
                }
      }


fclose(salida);

printf("Introducir nombre fichero con distancias hamming : ");
scanf("%s", nombre_fichero_distancias_hamming); //es el fichero de salida anterior

distancias_hamming=fopen(nombre_fichero_distancias_hamming, "r");

printf("Introducir nombre fichero para las frecuencias de cada distancia hamming : ");
scanf("%s", nombre_fichero_salida_frecuencias);

frecuencias_salida=fopen(nombre_fichero_salida_frecuencias, "w");

for(l=0; l<combinaciones; l++)
              {
              fscanf(salida, "%d", &distancia_hamming_leida);
              frecuencias[l]=distancia_hamming_leida;
              }

 for(distancia_hamming=0; distancia_hamming<N_digitos+1; distancia_hamming++)
     {
              for (l=0;l<combinaciones;l++)
          {
              if(frecuencias[l]==distancia_hamming)
                  numero_repeticiones++;
          }
            fprintf(frecuencias_salida, "%d\n", numero_repeticiones);
            numero_repeticiones=0;
     }
}
```

Este otro código lee el fichero en el que se encuentran las Distancias Hamming y obtiene la media y la varianza de los valores.

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

#define N_claves 20
#define N_digitos 99
#define combinaciones 190

int main ()
{
    FILE *distancias, *estadistica;
    char nombre_fichero_distancias_hamming[100], nombre_fichero_estadistica[100];
    int i, distancia_hamming_leida, distancias_hamming[combinaciones], suma=0;
    double media=0, varianza=0;


    printf("Introducir nombre fichero distancias hamming: ");
    scanf("%s", nombre_fichero_distancias_hamming);

    distancias=fopen(nombre_fichero_distancias_hamming, "r");

    if (distancias==NULL)
    {
        printf("Error al abrir el fichero\n");
        return 0;
    }

    printf("Introducir nombre fichero estadistica: ");
    scanf("%s", nombre_fichero_estadistica);

    estadistica=fopen(nombre_fichero_estadistica, "w");

    if (estadistica==NULL)
    {
        printf("Error al abrir el fichero\n");
        return 0;
    }

for(i=0; i<combinaciones; i++)
            {
            fscanf(distancias, "%d", &distancia_hamming_leida);
            distancias_hamming[i]=distancia_hamming_leida;
            }

 for(i=0; i<combinaciones; i++)
            suma=suma+distancias_hamming[i];

            media=1.0*suma/(combinaciones);


for(i=0; i<combinaciones; i++)
    varianza+=(distancias_hamming[i]-media)*(distancias_hamming[i]-media);
    varianza=varianza/(combinaciones-1); //varianza muestral sesgada

fprintf(estadistica, "%f %f", media, varianza);

}
```

Por último, el tercer código lee un fichero que contiene claves y devuelve la clave más probable comparando si en cada dígito es más probable que aparezca un 0 o un 1.

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

#define N_claves 20
#define N_digitos 99
#define combinaciones 190

int main ()
{
    FILE *claves, *salida;
    char nombre_fichero_claves[100], nombre_fichero_salida[100];
    int matriz[N_claves][N_digitos], i,j, digito_leido, suma_ceros=0, suma_unos=0;


    printf("Introducir nombre fichero claves: ");
    scanf("%s", nombre_fichero_claves);

    claves=fopen(nombre_fichero_claves, "r");

    if (claves==NULL)
    {
        printf("Error al abrir el fichero\n");
        return 0;
    }

    printf("Introducir nombre fichero salida: ");
    scanf("%s", nombre_fichero_salida);

    salida=fopen(nombre_fichero_salida, "wt");

    if (salida==NULL)
    {
        printf("Error al abrir el fichero\n");
        return 0;
    }


  for(i=0; i<N_claves; i++)
  {
      for(j=0; j<N_digitos; j++)
          {
          fscanf(claves, "%d", &digito_leido);
          matriz[i][j]=digito_leido;
          }
  }

  for (j=0; j<N_digitos; j++)
  {
      for(i=0;i<N_claves; i++)
      {
          if(matriz[i][j]==0)
              suma_ceros=suma_ceros+1;
          if(matriz[i][j]==1)
              suma_unos=suma_unos+1;
      }
```

```
 if (suma_ceros>suma_unos) fprintf(salida, "%d", 0);
if (suma_unos>suma_ceros) fprintf(salida, "%d", 1);
if (suma_unos==suma_ceros) fprintf(salida, "%c", 'x');
fprintf(salida, "   ");
    suma_ceros=0;
    suma_unos=0;
}
```