UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

**Instituto Universitario de Microelectrónica Aplicada**

**Sistemas de información y Comunicaciones**

# Máster en
# Tecnologías de Telecomunicación


## Trabajo Fin de Máster

## Análisis y diseño de aceleradores hardware sobre SoC basados en FPGA orientados a aplicaciones de seguridad en Internet de las Cosas

Autor: D. Yúbal Barrios Alfaro

Tutor(es): Dr. Pedro Pérez Carballo
Dr. Antonio Núñez Ordóñez

Fecha: julio de 2017

IUMA

**t** +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira

**f** +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

**Instituto Universitario de Microelectrónica Aplicada**

**Sistemas de información y Comunicaciones**

# Máster en
# Tecnologías de Telecomunicación


# Trabajo Fin de Máster

## Análisis y diseño de aceleradores hardware sobre SoC basados en FPGA orientados a aplicaciones de seguridad en Internet de las Cosas

## HOJA DE FIRMAS

| | | | |
|---|---|---|---|
| **Alumno/a:** | Yúbal Barrios Alfaro | Fdo.: | BARRIOS ALFARO, YUBAL (FIRMA)<br>Firmado digitalmente por BARRIOS ALFARO, YUBAL (FIRMA)<br>Nombre de reconocimiento (DN): c=ES, serialNumber=45779183Y, sn=BARRIOS, givenName=YUBAL, cn=BARRIOS ALFARO, YUBAL (FIRMA)<br>Fecha: 2017.07.19 12:35:21 +01'00' |
| **Tutor/a:** | Pedro Pérez Carballo | Fdo.: | Firmado por PEREZ CARBALLO, PEDRO FRANCISCO (FIRMA) el día 19/07/2017 con un certificado emitido por AC DNIE 001 |
| **Tutor/a:** | Antonio Núñez Ordóñez | Fdo.: | NUÑEZ ORDOÑEZ ANTONIO - 50278395G<br>Firmado digitalmente por NUÑEZ ORDOÑEZ ANTONIO - 50278395G<br>Fecha: 2017.07.18 23:00:25 +01'00' |

**Fecha: julio de 2017**

IUMA

**t** +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira

**f** +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

**Instituto Universitario de Microelectrónica Aplicada**

**Sistemas de información y Comunicaciones**

# Máster en
# Tecnologías de Telecomunicación


# Trabajo Fin de Máster

## Análisis y diseño de aceleradores hardware sobre SoC basados en FPGA orientados a aplicaciones de seguridad en Internet de las Cosas

## HOJA DE EVALUACIÓN

**Calificación:** .....

**Presidente** Fdo.:

**Secretario** Fdo.:

**Vocal** Fdo.:

**Fecha: julio de 2017**

IUMA

**t** +34 928 451 086 | iuma@iuma.ulpgc.es | Campus Universitario de Tafira

**f** +34 928 451 083 | www.iuma.ulpgc.es | 35017 Las Palmas de Gran Canaria

# AGRADECIMIENTOS

En primer lugar, me gustaría comenzar dando las gracias a mis tutores, Pedro Pérez Carballo y Antonio Núñez Ordóñez, por la confianza depositada en mí para acometer proyectos de investigación dentro de la división SICAD del IUMA como el que se expone en el presente documento, así como por su dedicación, tiempo y esfuerzo para lograr que este trabajo culminara de forma positiva.

Asimismo, veo preciso resaltar a mis compañeros de grupo de trabajo, especialmente a Julian Spahr, por el buen ambiente creado en el laboratorio durante este tiempo y a la colaboración prestada con el objetivo de sacar adelante nuestros proyectos, contribuyendo a hacer de estos meses una etapa única e inolvidable.

Por último, es imposible para mí finalizar este apartado sin dar las gracias a las personas más importantes de mi vida, mi familia, por ser mi apoyo incondicional sobre todo cuando más en contra se han dado las circunstancias.

# RESUMEN

En este trabajo se ha realizado el modelado, diseño e implementación sobre un MPSoC programable basado en FPGA de un nodo *Fog* orientado a asegurar las comunicaciones en aplicaciones de Internet de las Cosas. Asimismo, el diseño se ha llevado a cabo teniendo en cuenta los requisitos de latencia y consumo de potencia del paradigma de *Fog Computing*.

El citado sistema está compuesto por un bloque IP encargado de realizar la tarea de descifrado de la información recibida por el nodo y por un filtro basado en una estructura *Counting Bloom Filter* que analiza la cabecera de los paquetes Ethernet recibidos a nivel de las capas de red y transporte para detectar posibles anomalías que puedan influir en un incorrecto funcionamiento del sistema. El proceso de diseño e implementación de cada uno de los bloques IP citados se describe en detalle, tanto a nivel *hardware* como *software*.

Para alcanzar el citado objetivo, se hace necesario un estudio previo tanto de las especificaciones que debe cumplir un dispositivo incluido en una arquitectura IoT como la que tiene como fin este proyecto, así como de las aplicaciones de seguridad disponibles que se pueden implementar en un MPSoC, haciendo balance sobre las ventajas e inconvenientes que presentan. Asimismo, se detalla la metodología de diseño basada en síntesis de alto nivel empleada y las herramientas que se han decidido utilizar para completar de forma satisfactoria todas las etapas del flujo de diseño.

Finalmente, se presenta el banco de validación empleado para comprobar la funcionalidad del sistema completo y se analizan los resultados obtenidos en términos de latencia, consumo de potencia y área consumida por el diseño, comparándolos con otros trabajos similares disponibles en el estado del arte y concluyendo que la solución alcanzada cumple con los requisitos necesarios para trabajar a velocidades de Gigabit por segundo.

# ABSTRACT

This work summarizes the design of a Fog node and its implementation over a programmable and FPGA-based MPSoC for secure communications in an IoT environment. Besides, the design has been made keeping in mind the Fog Computing latency and power consumption requirements.

This system is comprised of an IP block that makes the decryption of the Ethernet frames received by the Fog node and a second IP, a Counting Bloom Filter that analyses the Ethernet packet header at a network and transport level layer to detect potential threats that can jeopardize the system. The design and implementation process of these IP blocks is described meticulously, for both the hardware and software domains.

To reach this goal, it is necessary to study previously the device specifications for its integration in IoT architecture and the available security applications that can be implemented in an MPSoC, evaluating their features. We also cover the High-Level Synthesis design methodology and the software tools used during this project.

Afterwards, the validation testbench used to verify the system's performance is presented and the final results are analysed in terms of latency, power consumption and area, comparing them with similar works presented in the State of the Art. Finally, we can conclude that the presented solution satisfies the requirements to work at Gigabit per second speeds.

# Tabla de contenido

# Índice de figuras

# Índice de tablas

# ACRÓNIMOS

| | |
|---|---|
| 6LoWPAN | *IPv6 over Low power Wireless Personal Area Networks* |
| AES | *Advanced Encryption Standard* |
| AH | *Authentication Header* |
| AMBA | *Advanced Microcontroller Bus Architecture* |
| ARM | *Advanced RISC Machine* |
| ASIC | *Application-Specific Integrated Circuit* |
| AXI | *Advanced eXtensible Interface* |
| BCA | *Bus Cycle Accurate* |
| BF | *Bloom Filter* |
| BRAM | *Block Random Access Memory* |
| BSP | *Board Support Package* |
| CABA | *Cycle Accurate Bit Accurate Modelling* |
| CBF | *Counting Bloom Filter* |
| CPU | *Central Processing Unit* |
| DDoS | *Distributed Denial of Service* |
| DOI | *Domain Of Interpretation* |
| DSP | *Digital Signal Processor* |
| ECC | *Elliptic Curves Cryptography* |
| ESL | *Electronic System Level* |
| ESP | *Encapsulating Security Payload* |
| FF | *Flip-Flop* |
| FIFO | *First In, First Out* |
| FPGA | *Field Programmable Gate Array* |
| FU | *Factor de Utilización* |
| HDL | *Hardware Description Language* |
| HLS | *High-Level Synthesis* |
| HMAC | *keyed-Hash Message Authentication Code* |
| HPC | *High-Performance Computing* |
| IEEE | *Institute of Electrical and Electronic Engineers* |
| ILA | *Integrated Logic Analyzer* |

| | |
|---|---|
| IoT | *Internet of Things* |
| IP | *Internet Protocol (OSI model layer) or Intellectual Property (for blocks design)* |
| IPSec | *Internet Protocol Security* |
| LUT | *Look-Up Table* |
| M2M | *Machine-To-Machine* |
| MPSoC | *MultiProcessor System-on-Chip* |
| NaCl | *Networking and Cryptography library* |
| NPF | *Network Packet Filter* |
| NRE | *Non-Recurring Engineering* |
| NSA | *U.S. National Security Agency* |
| OS | *Operating System* |
| OSI | *Open System Interconnection* |
| PL | *Programmable Logic* |
| PLL | *Phase Locked Loop* |
| PPA | *Power Performance Area* |
| PS | *Processing System* |
| RTL | *Register Transfer Level* |
| SD | *Secure Digital* |
| SDSoC | *Software-Defined System-on-Chip* |
| SoC | *System-on-Chip* |
| SPI | *Security Parameters Index* |
| SPN | *Substitution-Permutation Network* |
| SSH | *Secure SHell* |
| TCP | *Transmission Control Protocol* |
| TFM | *Trabajo Fin de Máster* |
| TTM | *Time-To-Market* |
| UDP | *User Datagram Protocol* |
| VPN | *Virtual Private Network* |
| WNS | *Worst Negative Slack* |

# Capítulo 1. Introducción

## 1.1. Antecedentes

### 1.1.1. Estado actual de Internet tras la irrupción del Internet de las Cosas

En la actualidad, el número de dispositivos que se encuentran conectados a Internet se incrementa de forma continua a un ritmo del 8% anual según los datos aportados por el gigante americano de las telecomunicaciones Cisco, en su previsión para el periodo 2016-2021 [1]. Gran parte de este crecimiento se debe, sin duda, a la irrupción del concepto de Internet de las Cosas, paradigma tecnológico que supone la conexión a la red de prácticamente cualquier dispositivo imaginable empleado en nuestra vida cotidiana (desde frigoríficos a sistemas de riego inteligentes), con el fin de facilitar el día a día a la sociedad. Según las predicciones de diversas empresas especializadas, la cifra de dispositivos con conexión a Internet en el año 2020 será cercana a los 50 mil millones (a más de 6 dispositivos por persona) [2][3], situación que se refleja en la Figura 1, en la que los dispositivos con comunicación *machine-to-machine* (M2M) adquieren especial relevancia, con una cuota prevista de mercado en 2021 del 29%, solo por detrás de los *smartphones*, dispositivo electrónico dominante en el mercado actual [1].

Asimismo, según la predicción realizada por IC Insights, se estima que las ventas de dispositivos semiconductores vinculados a aplicaciones de IoT alcanzarán los 31,1 mil millones de dólares en el año 2020, con un incremento de casi el 10% respecto a la estimación para el año 2017 (21,3 mil millones de dólares) [4]. Esta información proporciona un dato más del auge que presenta y presentará esta tecnología en el próximo lustro.

Por tanto, a la transmisión y almacenamiento de grandes cantidades de información en Internet, derivadas de las actividades habituales del usuario tanto de carácter personal (redes sociales, correo electrónico, transacciones bancarias, etc.) como corporativa [5], hay que añadir los dispositivos asociados a Internet de las Cosas, que también se encuentran conectadas a la red en todo momento y generando continuamente tráfico de datos. Este paradigma promueve, asimismo, el desarrollo de diferentes niveles de objetos inteligentes, como pueden ser *Smart grids* [6], *Smart Cities* [7] , etc.

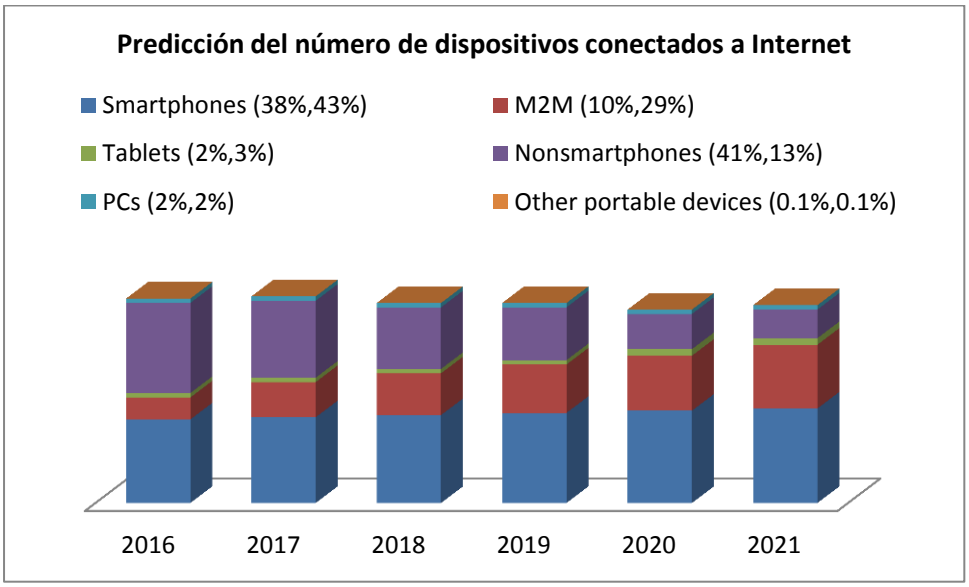



**Figura 1. Número de dispositivos conectados a Internet por tipos. Adaptada de [1].**

## 1.1.2. Cibercriminalidad

El gran volumen de datos que se genera actualmente en la red y que, tal como se ha comentado en el apartado anterior, sufrirá un incremento considerable a corto plazo, presenta un claro interés para los cibercriminales, que tienen acceso ahora a mayor cantidad de información del usuario más allá de la que comúnmente se podían encontrar en Internet hasta ahora (información personal, laboral o bancaria), como pueden ser registros médicos o hábitos vitales [8], derivados de la introducción de dispositivos IoT en ámbitos tales como la medicina, la industria o la domótica, entre otros.

De hecho, tal como se muestra en la Figura 2, el cibercrimen, que incluye prácticas tales como *phising*, *malware* o *spam* con el único fin de obtener información privada del usuario con la que posteriormente hacer negocio, se trata de la mayor ciberamenaza en la actualidad con casi un 67% sobre el total [9], porcentaje que presumiblemente continuará

en aumento en los próximos años situándose, según las previsiones, sobre el 90% en el año 2020.

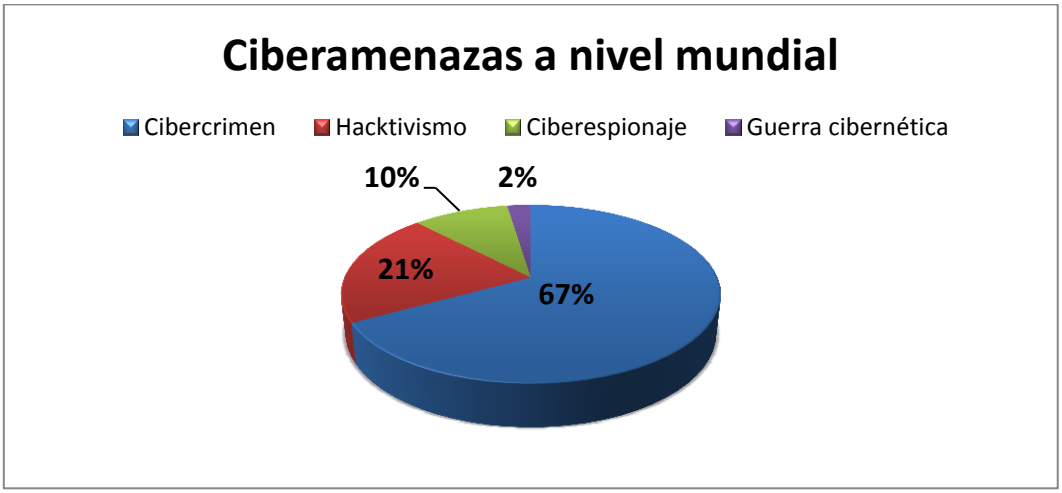



**Figura 2. Ciberamenazas por tipos. Adaptada de [9].**

Como ejemplos de esta actividad delictiva en Internet, cabe resaltar el caso del servidor DNS Dyn, objeto de un ciberataque de denegación de servicio distribuido (DDoS) en octubre de 2016 [10] mediante el envío masivo de paquetes desde dispositivos IoT, principalmente cámaras IP, dejando inservible durante un largo periodo temporal grandes servicios de Internet como Twitter y Spotify. La cantidad de ataques de este tipo sigue en continuo crecimiento, habiendo tenido también una repercusión importante la retirada por parte de la empresa automovilística estadounidense Chrysler de un total de 1.4 millones de vehículos durante el verano de 2015 al haberse dado casos de control remoto de los mismos por parte de personas no autorizadas [11], o el ataque DDoS sufrido en noviembre de 2016 en la ciudad finlandesa de Lappeenranta, por el cual dos edificios automatizados vieron mermados sus sistemas de calefacción [12].

Con ello, se ha puesto en evidencia que los dispositivos actuales de IoT presentan carencias graves es aspectos de seguridad [13], lo que da lugar a su control por parte de cibercriminales que pueden llevar a cabo acciones delictivas con la información capturada. Afortunadamente, cada vez mayor número de voces autorizadas en el ámbito de la ciberseguridad empiezan a ejercer presión sobre los organismos internacionales, con el fin de hacer frente al riesgo que supone contar con una infraestructura de IoT insegura y que podría provocar un atentado contra la privacidad de los usuarios y contra la propia seguridad nacional e internacional; de esta forma, se está insistiendo en la declaración de una ley que

asegure ciertos criterios de seguridad por parte de las empresas encargadas de la fabricación de los dispositivos IoT. En palabras de Bruce Schneier, considerado un gurú en el ámbito de la ciberseguridad y profesor de políticas de seguridad en la Universidad de Stanford, recogidas por la web MIT Technology Review en diciembre de 2016 [14], *"el ataque DDoS al servidor Dyn es sólo una pequeña parte del riesgo que supone contar con una infraestructura de IoT insegura ya que, en la actualidad, este tipo de dispositivos se encuentran en entornos tales como hospitales o en el control de ascensores y de aparatos de monitorización de bebés, por lo que no es difícil imaginar una catástrofe de mayores dimensiones. El principal problema en este sentido radica en el hecho de que no existe ningún tipo de ley o institución que aplique algún tipo de normativa relativa a la seguridad a los fabricantes de dispositivos, los cuales anteponen otras muchas características a la vulnerabilidad frente a ciberataques"*.

En este sentido, cobra especial importancia el control de lo que se denomina la *Deep Web* ya que, desde esta parte menos conocida de Internet, es desde la que se generan la mayoría de actividades cibercriminales que tienen lugar *online*. A modo de resumen, la *Deep Web* es aquella parte de Internet que no es visible a los usuarios comunes ya que el material, información y sitios web que aloja no se encuentran indexados por los algoritmos de los motores de búsqueda existentes como Google, Bing o Yahoo. En la actualidad, se estima que su tamaño es de 500 veces la parte visible del Internet que conocemos [15].

### 1.1.3. Concepto de Fog Computing

De este modo, es fácil suponer que esta tecnología requiere de cambios en las técnicas de conectividad, latencia y seguridad empleadas hasta ahora en la gestión de redes. Cisco introduce en el año 2016 el concepto de *Fog* o *Edge Computing* [16], consistente en llevar parte de los recursos de la nube a una capa distribuida cercana a los dispositivos IoT, con el objeto de reducir la latencia de análisis de la información crítica (aplicaciones médicas, control de plantas industriales, etc.). Con ello se proporciona una capa de seguridad dedicada a estos dispositivos, de forma más eficiente que si dichas tareas se realizan directamente en centros de procesamiento en la nube, tal como ocurre de forma común en la actualidad [17]. En la Figura 3 se observa la arquitectura de capas de procesamiento resultantes, permitiendo que la capa *Fog* opere en una zona intermedia entre los dispositivos de campo y los grandes centros de HPC.

De este modo, se consigue simplificar los dispositivos finales, sin necesidades de procesamiento o almacenamiento de datos (procedimientos llevados a cabo por el *Fog Node*) y conllevando con ello un menor consumo energético y una mayor vida útil, siendo su única misión la recogida y envío de información de interés [18][19].

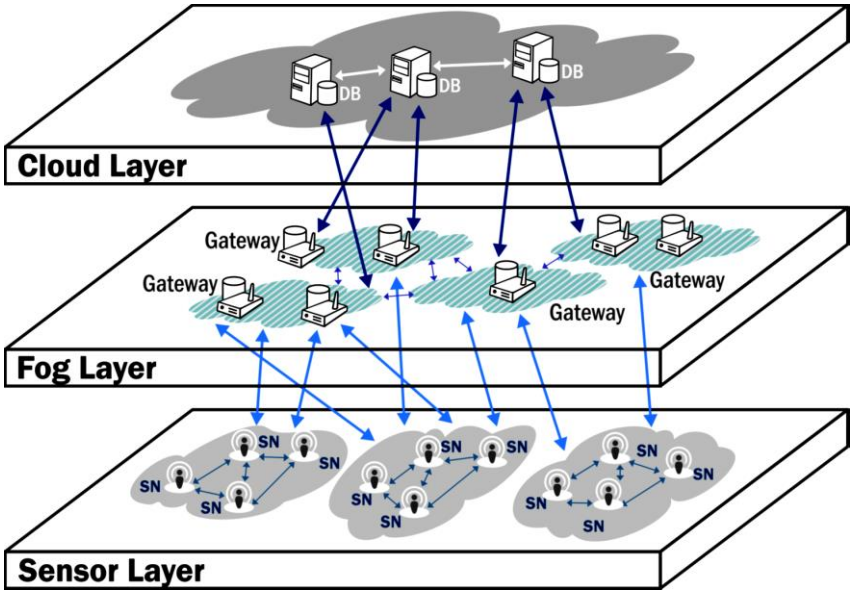



**Figura 3. Modelo Fog Computing para IoT [19].**

Por otro lado, en la Tabla 1 se resumen algunas diferencias entre las capas de procesamiento en la nube y la capa *Fog* en función de determinados parámetros clave, como son la latencia de respuesta o la seguridad:

**Tabla 1. Comparativa entre la nube y la capa Fog**

| Parámetro | Cloud Computing | Fog Computing |
|---|---|---|
| **Latencia** | Alta | Baja |
| **Localización** | Centros de HPC | Cerca de los dispositivos IoT |
| **Distribución** | Centralizada | Distribuida |
| **Seguridad** | Dependientes del proveedor de los servicios | Políticas de privacidad personalizadas |
| **Número de nodos** | Pocos | Bastantes |
| **Soporte para movilidad** | Limitado | Soporte absoluto |
| **Aplicaciones de tiempo real** | Soportadas pero difíciles de conseguir | Soportadas |
| **Tipo de conexión final** | Cableada, Gigabit Ethernet | *Wireless* |

En este sentido, las líneas de investigación se están centrando en desarrollar esta capa intermedia de procesamiento de información para aquellas arquitecturas de sensores y actuadores guiados por tecnologías inalámbricas (Wi-Fi, Bluetooth y Bluetooth LE, ZigBee, etc.) cuyas aplicaciones requieren de un gran compromiso en términos de margen de respuesta y seguridad se refiere, como pueden ser las *Smart Grids*, *Smart Cities*, soluciones aplicadas al entorno médico o el desarrollo de vehículos autónomos [20].

Sin embargo, al tratarse de una tecnología y un paradigma relativamente reciente, existen aún brechas existentes sobre todo en el ámbito de la seguridad [13][17], por lo que se hace necesario el desarrollo de sistemas que permitan asegurar la privacidad de la información transmitida, así como evitar ciberataques bastante comunes, como pueden ser ataques DDoS, *Man-in-the-Middle* o *Intrusion Detection* [21][22].

Los avances en la capacidad de integración de la tecnología facilitan la incorporación de aceleradores *hardware* especializados, lo que implica una necesaria reorganización del *hardware* empleado (se pasan funciones de un procesador de propósito general a los aceleradores), así como de los algoritmos *software* de seguridad y tratamiento de la información (acercándolos a facilitar una implementación *harware friendly*).

Estos aceleradores *hardware* permiten mejorar las prestaciones en cuanto a latencia de análisis y seguridad se refiere, sin descuidar la reducción del consumo energético (los nodos *Fog* deben trabajar de forma autónoma durante un largo periodo de tiempo) y del área física que ocupa el dispositivo. En este sentido, se considera que las estructuras que incluyen conjuntamente núcleos procesadores y dispositivos programables (SoCs heterogéneos) en un único dispositivo representan una solución eficiente para dicho problema, permitiendo una alta capacidad de cómputo y una reducción significativa de la potencia global consumida por el sistema.

### 1.1.4. Empleo de aceleradores hardware

En los tiempos actuales, la capacidad de integración es tal en el proceso de diseño y fabricación de circuitos en chip que es posible incluir en un mismo dado tanto núcleos procesadores que cumplan con las necesidades de cómputo de una aplicación concreta, como dispositivos programables como son las FPGAs que favorezcan la reconfiguración del sistema.

Si nos centramos en la flexibilidad del sistema electrónico, las FPGAs representan una opción a tener en cuenta ya que, al tratarse de dispositivos reprogramables y reconfigurables, permiten incluir nuevas funciones o añadir características adaptadas a posibles condicionantes o situaciones adversas que puedan ir surgiendo con el transcurrir del tiempo.

Asimismo, se trata de un dispositivo óptimo para aplicaciones que requieren un reducido consumo de potencia, como pueden ser el mencionado IoT, la domótica o la industria aeroespacial. Sin embargo, este consumo de potencia va en aumento de forma proporcional con las prestaciones del sistema, por lo que se hace necesario alcanzar un compromiso entre ambos parámetros [23].

En este sentido hay que indicar también que, con el avance en las tecnologías de integración en chip actuales, la potencia estática domina, por debajo de los 90 nm de longitud del canal de los MOSFET, el consumo de energía del SoC (Figura 4). Dicha potencia se deriva de las corrientes de fuga que tienen lugar en los transistores que incluye el PL, siendo esta corriente mayor cuanto menor es la longitud del canal de los transistores [24]. De esta forma, para poder reducir el consumo de potencia del sistema, el diseñador debe centrarse en reducir la potencia dinámica, la única que puede gestionar y que resulta tanto de la actividad que realiza el SoC como de la tensión de alimentación la cual, por lo general, se reduce con cada nuevo proceso de fabricación que surge en la tecnología de integración en chip.

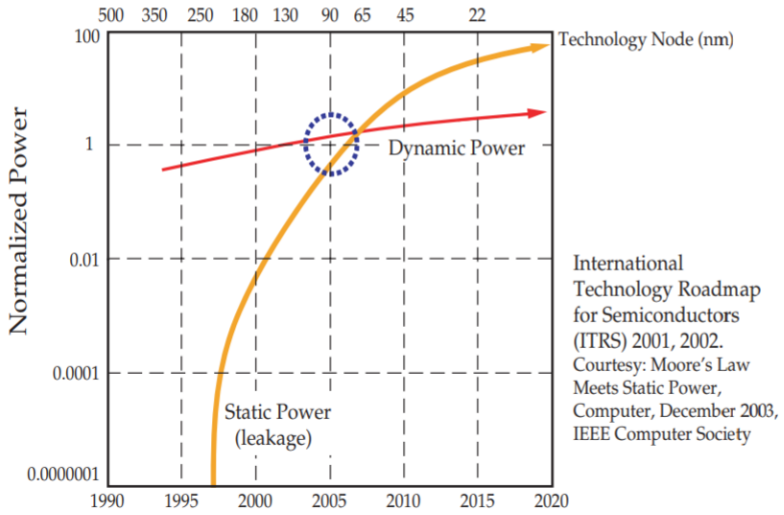



**Figura 4. Relación entre potencia estática y dinámica según el proceso de fabricación [24].**

Integrando los bloques FPGA con los núcleos procesadores mencionados, se obtiene lo que hoy en día conocemos como un SoC programable que, por sus prestaciones, potencia, facilidad de diseño y reconfigurabilidad, están siendo empleados en la actualidad en numerosas aplicaciones, tales como procesamiento de audio y vídeo, o sistemas de seguridad en las comunicaciones digitales [25], por citar algunos ejemplos.

Este estado de madurez de la tecnología ha llamado la atención de las grandes empresas del sector digital, como son IBM, Microsoft, Amazon o Facebook [26], que demandan la integración de este tipo de aceleradores en sus servidores de cómputo con el fin de aumentar las prestaciones de los mismos, amén de una reducción del considerable consumo de potencia que se deriva de estos grandes centros de procesamiento [27]. Por otro lado, Intel, gigante de la industria de semiconductores y líder en el sector de microprocesadores para PCs, adquirió en 2016 Altera, segundo líder mundial en fabricación y venta de FPGAs por detrás de Xilinx, poniendo en el mercado durante el mismo año SoCs programables compuestos por procesadores Xeon y FPGAs de las familias Arria y Stratix, destinados a aplicaciones de procesamiento en la nube [28].

En la Figura 5 se presenta una comparación entre este tipo de dispositivos con otros existentes en el mercado, como pueden ser procesadores de propósito general (GPP), GPUs (*Graphics Processing Units*), DSPs (*Digital Signal Proccessors*) o ASICs (*Application Specific Integrated Circuits*), en lo que a flexibilidad en la programación y a eficiencia se refiere.

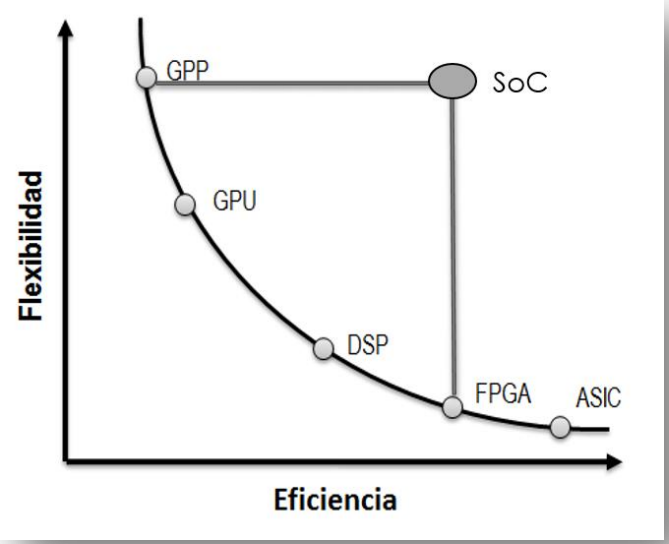



**Figura 5. Relación entre flexibilidad y eficiencia entre soluciones de implementación.**

### 1.1.5. Dispositivo Zynq de Xilinx

Además de las soluciones SoC proporcionadas por Intel FPGA (Altera), en la actualidad podemos encontrar en el mercado plataformas de este tipo (*hard core* + FPGA) pertenecientes a otras empresas relacionadas con el ámbito de los dispositivos programables. En nuestro caso concreto, nos centraremos en las opciones que proporciona Xilinx, empresa líder en el ámbito de los dispositivos lógicos programables.

La familia de dispositivos SoCs Zynq de Xilinx integran una FPGA de la serie 7 (Artix o Kintex según el dispositivo) junto con una unidad de procesamiento formada por dos núcleos ARM Cortex-A9 capaces de alcanzar altas tasas de cómputo. Además, incluye una infraestructura de buses AMBA AXI que comunica los módulos de procesamiento con la FPGA (bloque PL o de lógica programable), facilitando su comunicación con interfaces externas y pudiendo dar origen a sistemas *hardware* de mayor complejidad (Figura 6) [29].

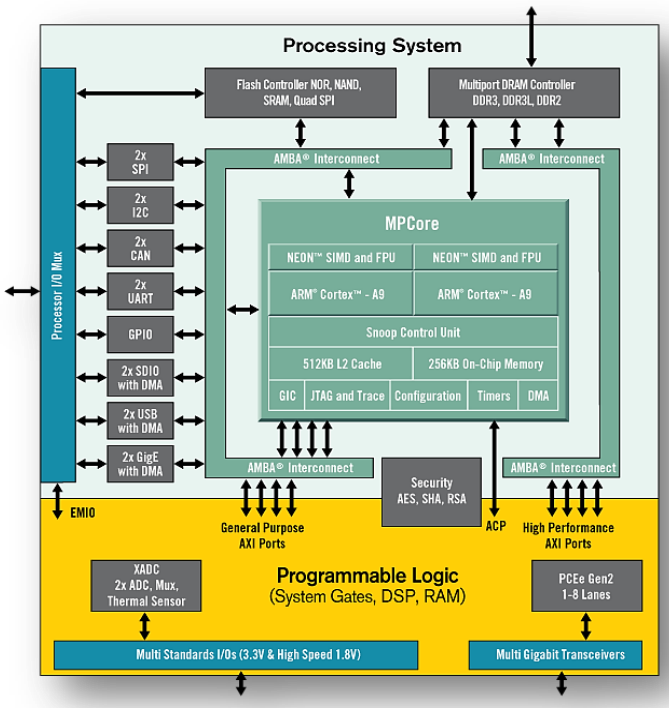



**Figura 6. Estructura de la plataforma Zynq de Xilinx [29].**

Recientemente, Xilinx ha puesto en el mercado la familia Zynq UltraScale+ que incorpora, en su caso, cuatro núcleos procesadores ARM Cortex-A53, dos núcleos Cortex-R5 para aplicaciones de tiempo real, una GPU Mali 400 e, incluso, un bloque decodificador de vídeo H.264-H.265 dedicado [30].

Para este proyecto en concreto, se ha decidido emplear un SoC de la familia Zynq, debido a que en el grupo de investigación se cuenta con una amplia experiencia en la realización de proyectos en los que tanto los dispositivos como las herramientas *software* de Xilinx han formado parte clave en la etapa de diseño e implementación. Asimismo, se empleará como plataforma física de validación una placa de desarrollo ZedBoard™ Zynq®-7000 ARM/FPGA SoC, que incluye un dispositivo Zynq XC7Z020-CLG484-1 EPP con los dos citados núcleos ARM Cortex-A9 de 32 bits y una FPGA Kintex de la serie 7, que dispone de los recursos tanto lógicos como de procesamiento suficientes para la implementación asociada al nodo *Fog* que se diseñará en este proyecto. En la Figura 7 se muestra el *layout* de dicha placa, resaltando sus principales componentes [31].

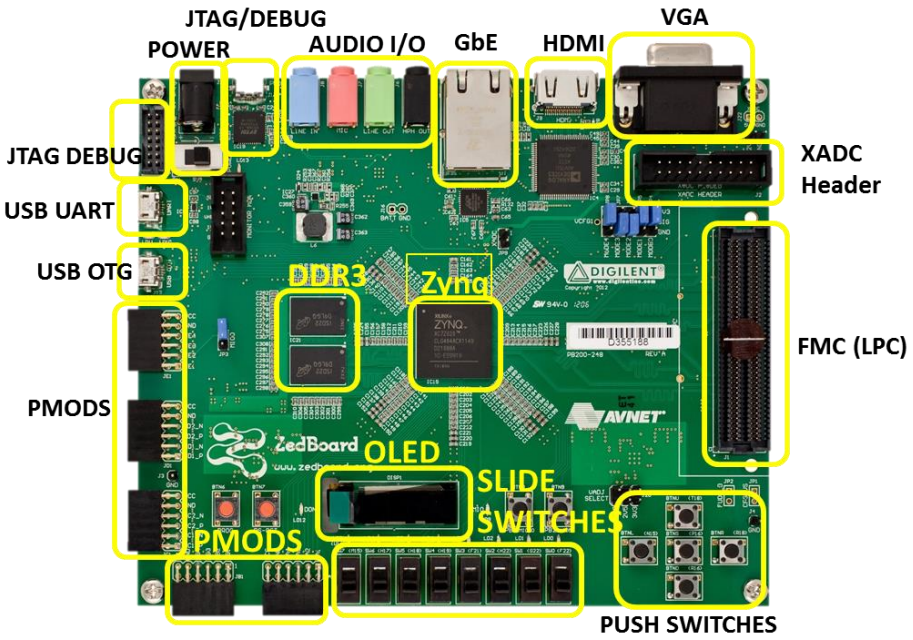



**Figura 7. *Layout* de la plataforma ZedBoard™.**

## 1.1.6. Flujo de diseño

El empleo de un MPSoC, como es el caso del dispositivo Zynq de Xilinx, precisa de una modificación del flujo de diseño tradicional en el ámbito de los sistemas electrónicos empotrados, siendo necesaria la introducción de nuevas etapas como son las que se mencionan seguidamente:

1) Análisis de la aplicación y partición *hardware/software* del problema, atendiendo a parámetros como las necesidades de cómputo de las tareas críticas a implementar o el consumo de memoria.

2) Diseño *hardware* del coprocesador, seleccionando la funcionalidad concreta que se requiere, así como las interfaces de comunicación necesarias.

3) Diseño e integración de la plataforma basada en IPs, donde la arquitectura de buses del sistema juega un papel clave, al llevar a cabo todas las comunicaciones entre los distintos módulos que conforman el SoC.

4) Diseño del *software* empotrado que gobernará el correcto funcionamiento del sistema, integración con el *hardware* y validación del sistema completo.

De esta forma, se obtiene un diseño cuya complejidad aumenta significativamente, al incluir etapas tanto en el dominio *hardware* como en el *software*.

Para dar soporte al desarrollo de sistemas electrónicos sobre la plataforma indicada, Xilinx ha desarrollado el entorno de diseño *Vivado® Design Suite* [32]. Este entorno incluye diferentes herramientas que cubren el flujo de diseño del sistema, desde la especificación algorítmica, su implementación y el desarrollo y depurado de la aplicación empotrada.

*Vivado High-Level Synthesis* (Vivado HLS) [33] parte de una descripción algorítmica en C/C++/SystemC del sistema, y crea la arquitectura a medida de los coprocesadores para la aplicación concreta, generando además las interfaces de comunicación estandarizadas para su integración con el resto del sistema (infraestructura de buses AMBA AXI, mapeado de memoria, etc.) [34].

El flujo de diseño continúa integrando dicho coprocesador a medida en la plataforma e implementando el sistema en la FPGA, para finalmente abordar el diseño del *software* empotrado y depurarlo *in situ* sobre la plataforma (Figura 8) [35]. Esta flexibilidad añadida al flujo de diseño permite la reconfiguración de la FPGA cuando sea preciso para cambiar la funcionalidad o la aplicación actual de forma transparente para el diseñador.

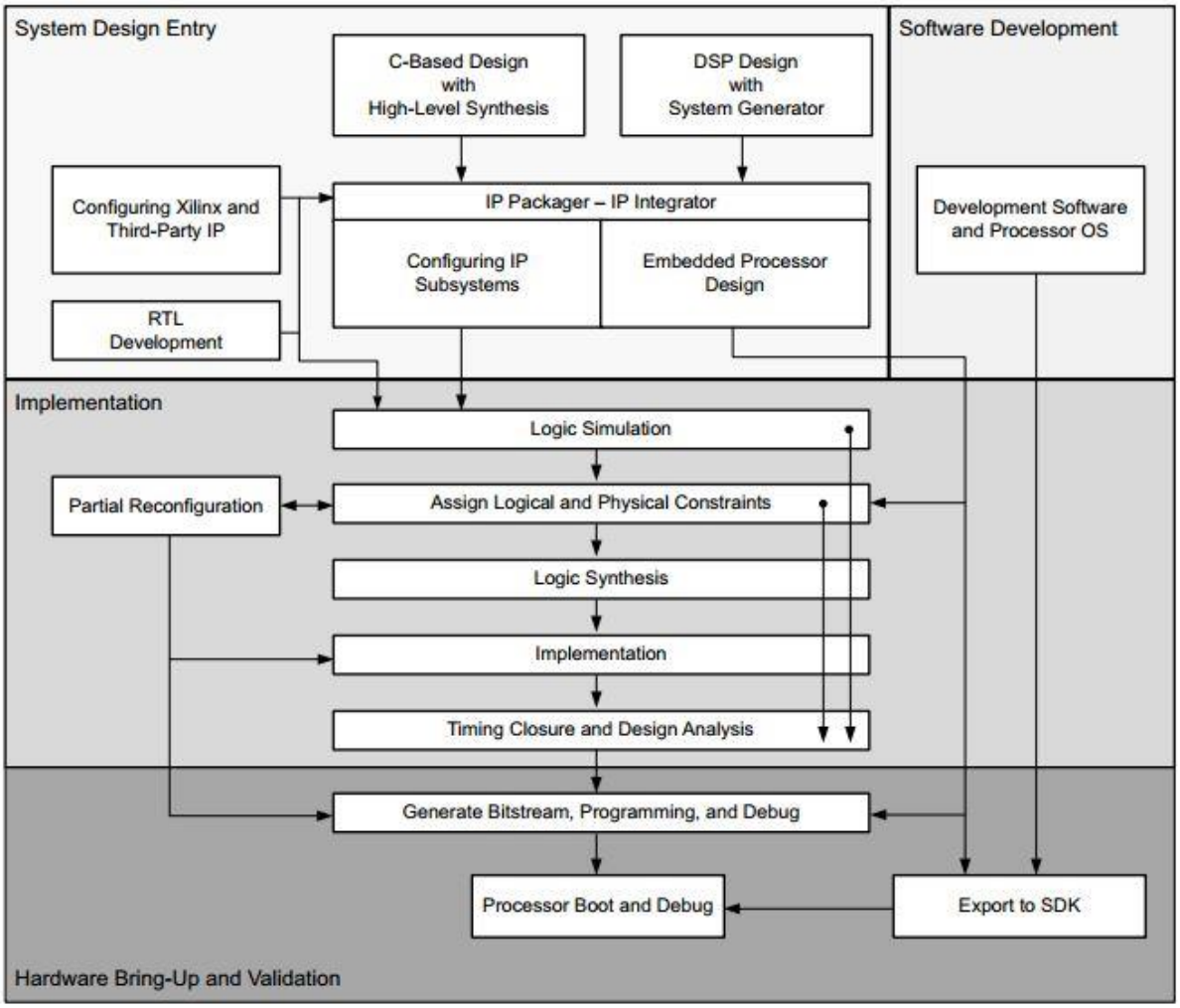



**Figura 8. Flujo de diseño de una plataforma empleando Xilinx Vivado [36].**

Asimismo, Xilinx ha introducido una herramienta de desarrollo denominada SDSoC™ (*Software-Defined System-on-Chip*). Esta herramienta integra un entorno de diseño completo para C/C++ basado en Eclipse, orientado al desarrollo de sistemas empotrados basados en SoC que soporta las familias Zynq®-7000 y Zynq UltraScale+. La característica principal de este nuevo entorno es que facilita la partición del sistema realizando un perfilado de la aplicación en alto nivel previo a realizar la partición *hardware/software* del sistema.

Una vez conocidas aquellas funciones que requieren de una mayor capacidad de cómputo, SDSoC proporciona una serie de herramientas que facilitan la creación del acelerador *hardware*, trasladándolas a la lógica programable disponible en la FPGA. Tomando esta partición como referencia, SDSoC realiza el flujo de diseño *hardware* equivalente al usado en Vivado HLS y Vivado IDE de forma automática y transparente, siendo posible por parte del diseñador, en caso de contar con las nociones básicas sobre

arquitecturas *hardware* (este entorno está principalmente enfocado a diseñadores *software* con escasa experiencia en *hardware*), modificar el sistema en cualquiera de las etapas de dicho flujo de trabajo. De forma adicional, tal como se observa en la Figura 9, permite iniciar el desarrollo del *software* empotrado en etapas iniciales del proceso de diseño, de forma contraria a lo que ocurría en el flujo de diseño tradicional de Vivado (Figura 8), en el cual el diseño del *software* se lleva a cabo prácticamente al final empleando Xilinx SDK (*Software Development Kit*) [37].

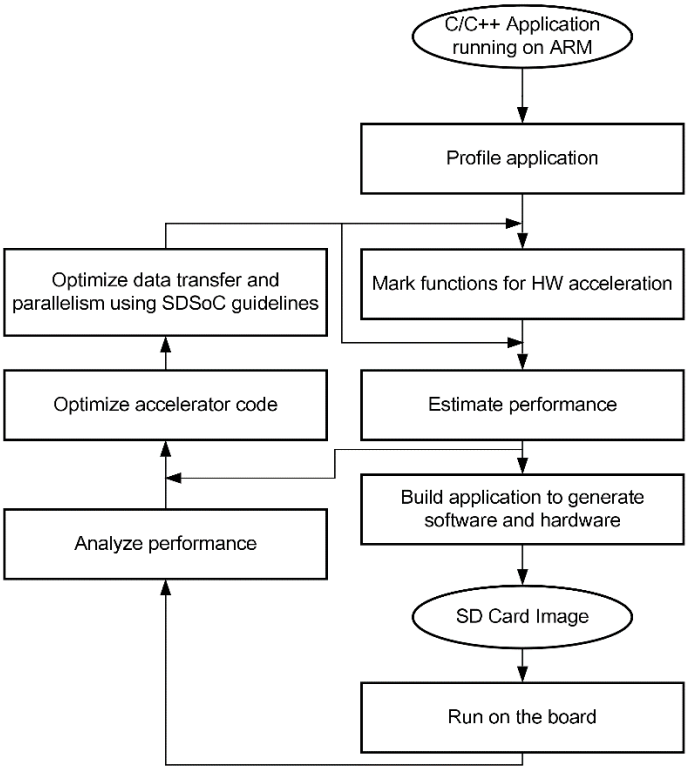



**Figura 9. Flujo de diseño SDSoC [37].**

## 1.2. Objetivos

A partir de lo explicado anteriormente, es de interés el desarrollo de un entorno seguro en el que el nodo *Fog* pueda trabajar con información crítica sin verse ésta comprometida por posibles ataques de terceros [17]. Partiendo de este requisito, como objetivo principal de este Trabajo Fin de Máster se ha fijado la implementación sobre un SoC reconfigurable Zynq de un acelerador *hardware* orientado a asegurar las comunicaciones en entornos de IoT.

Sin perder de vista este fin, se ha diseñado un sistema que permite el cifrado/descifrado de la información transmitida entre un equipo de control que pretende

acceder a un dispositivo IoT o a un rango determinado de ellos, y un nodo *Fog* que realiza las tareas de gestión y control de los mismos. Asimismo, se ha incluido un sistema de filtrado de paquetes de datos Ethernet que opera a un alto *throughput* (velocidades de Gigabit) y que permite rechazar conexiones desde equipos cuyos parámetros de red se identifiquen como una posible amenaza para el sistema o que intentan acceder a determinados puertos del nodo *Fog* no permitidos, actuando por tanto como sistema de autenticación. En la Figura 10 se muestra una visión global del sistema a implementar:

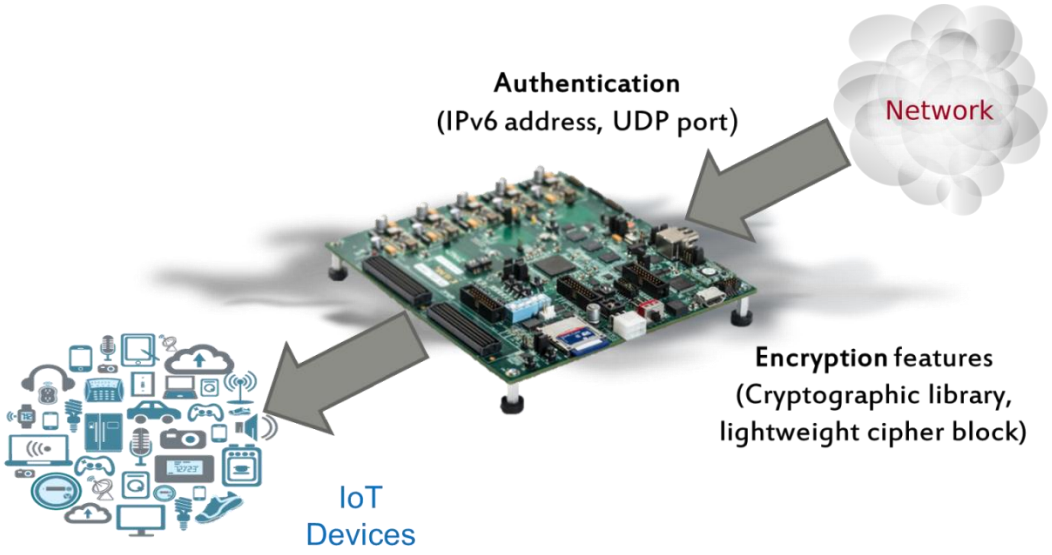



**Figura 10. Visión general del sistema.**

Al no tratarse el sentido de comunicación *host*-nodo *Fog* como un enlace crítico en cuanto a latencia se refiere, los retardos en la transmisión no constituyen un factor clave. Lo que se prima en este sistema es la seguridad, realzada mediante procesos de cifrado y descifrado de la información que se transfiere a través de las interfaces de red del dispositivo. Estas tareas de cifrado y descifrado de la información en la actualidad están basadas en un procesamiento en la nube. Ello provoca un considerable retardo en el envío y recepción de los datos. Por lo general, se considera que mientras más lento sea este proceso debido al mayor grado de complejidad que presenta el cifrado realizado, menos vulnerable será frente a accesos no autorizados por parte de posibles atacantes. Es en este aspecto en el que entran en juego los aceleradores *hardware*, tratando de reducir los tiempos de ejecución de este proceso sin verse afectada la robustez de los algoritmos empleados.

Haciendo referencia a la velocidad de las comunicaciones, en este trabajo se pretende cumplir, al menos, con los estándares IEEE 802.3z y IEEE 802.3ab para velocidades

de 1 Gigabit Ethernet sobre fibra óptica y par trenzado, respectivamente. Estos estándares aseguran la transmisión de datos a la velocidad indicada para una distancia máxima de 100 metros. Asimismo, se evaluará alcanzar las tasas de 2,5 y 5 Gbps definidas en el estándar IEEE 802.3bz [38].

Por su parte, el filtrado de paquetes de datos se realiza analizando directamente campos determinados de las cabeceras de los paquetes entrantes, de forma que el paquete en cuestión pueda ser descartado de forma previa al inicio del análisis del *payload*, dando lugar a una solución tipo *dataflow*, que evita tener que recibir el paquete completo para poder tomar una decisión sobre si será filtrado o no. El análisis de la cabecera se realiza atendiendo a dos parámetros distintos del *header* de un paquete de red: la dirección IP, en formato IPv6, del nodo origen (*blacklist*), y el puerto UDP del destino a través del cual se intenta establecer una conexión (*whitelist*).

Por tanto, para implementar el sistema descrito (Figura 11), se hace necesario el modelado e implementación de varios bloques IP que compondrán el diseño:

- Bloque de cifrado/descifrado. Realiza las operaciones vinculadas al cifrado y descifrado de la información a manejar por el sistema.
- Bloques de filtrado de paquetes de red. Se implementarán tantos bloques de filtrado como campos de la cabecera de los *paquetes* Ethernet se considere oportuno analizar (en este caso dos, para analizar los protocolos de las capas de red y transporte) para poder determinar si se descarta el paquete recibido o si se permite su acceso a aquellos dispositivos IoT gestionados por el nodo *Fog*.

Finalmente, indicar que el diseño planteado en este TFM se trata de la primera piedra para el desarrollo de un nodo *Fog* que incluya toda la funcionalidad requerida para trabajar de forma autónoma en un entorno real de IoT, tanto en términos de latencia, como de área y consumo de potencia. Por tanto, además de asegurar las comunicaciones en el entorno *cloud*-nodo *Fog*, se hará necesario ampliar la funcionalidad del nodo en futuros proyectos mediante el correcto establecimiento de enlaces con dispositivos IoT, tanto a nivel de seguridad como en cuanto a tratamiento y latencia de la información enviada, parámetro que en este caso sí es más relevante, al poder estar vinculados dichos dispositivos a

aplicaciones en los que la velocidad de respuesta es crítica (medicina, control industrial, *Smart Grids,* etc.).

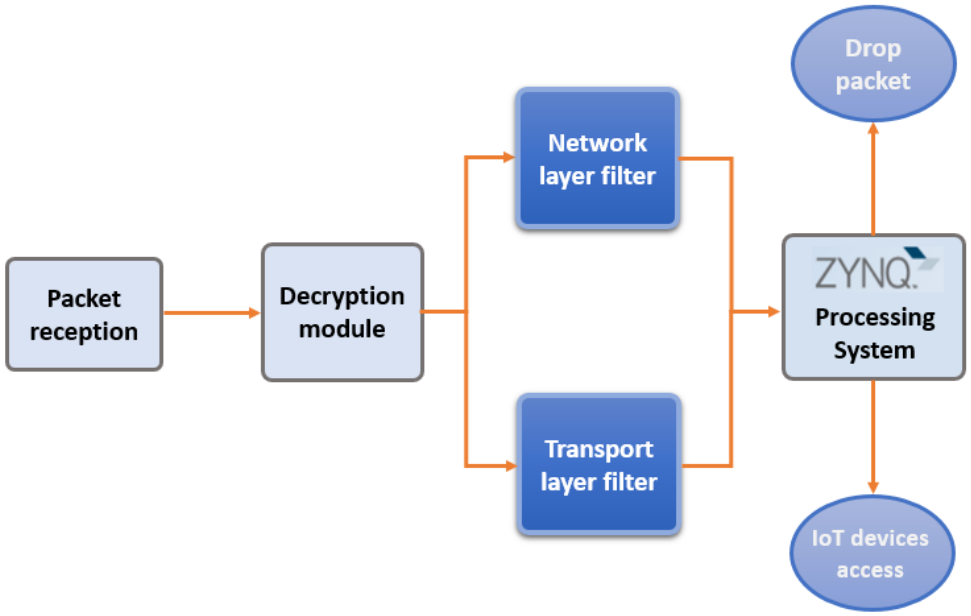



**Figura 11. Diagrama de bloques del sistema.**

## 1.3. Peticionario

La realización de este trabajo se realiza por petición de la División de Sistemas Industriales y CAD (SICAD), perteneciente al Instituto Universitario de Microelectrónica Aplicada (IUMA), dentro de la línea de investigación vinculada al modelado e implementación de sistemas integrados en chip.

Con este trabajo, se pretende realizar un acercamiento al empleo de aceleradores *hardware* en el ámbito de Internet de las Cosas, tecnología en auge actualmente y que requiere de sistemas eficientes en términos de prestaciones, área y consumo de potencia para satisfacer las demandas generadas.

## 1.4. Estructura del documento

Este documento se estructura como sigue. En este primer capítulo se indican los antecedentes que han motivado la realización de este trabajo, además de exponer los objetivos que se plantea alcanzar. En el capítulo 2 se describe la metodología aplicada para la realización del proyecto, además de justificar la elección de las decisiones tomadas para poder completar el proceso de diseño del sistema. Por su parte, en el capítulo 3 se aborda el

diseño e implementación de los bloques IP necesarios para realizar el filtrado de los paquetes Ethernet que recibe el *Fog Node* atendiendo a las políticas de acceso predefinidas, exponiendo sus características y cómo se rigen las comunicaciones entre ellos y el resto del sistema.

En el capítulo 4 se realiza un análisis de las alternativas de cifrado seleccionadas para evaluar cuál de ellas presenta mayores ventajas para una implementación *hardware*. El capítulo 5 detalla la integración de los bloques IP modelados en la plataforma junto con el *software* empotrado diseñado, para posteriormente verificar el funcionamiento del sistema completo. Por otro lado, en el capítulo 6 se realiza la validación *in situ* del sistema sobre el dispositivo físico, recogiendo los resultados obtenidos y realizando su comparación con trabajos realizados en la misma dirección que el aquí presentado.

Finalmente, en el capítulo 7 se citan las conclusiones alcanzadas tras finalizar la realización de este trabajo, así como las líneas futuras que se plantean para la posible ampliación y mejora de las prestaciones del sistema en futuros proyectos.

# Capítulo 2. Decisiones de diseño

## 2.1. Introducción

El objetivo de este capítulo es justificar las decisiones tomadas de forma previa y durante el proceso de diseño del sistema, así como realizar una aproximación a la metodología de diseño empleada para su modelado, implementación y verificación. Asimismo, se compararán las herramientas disponibles para las distintas etapas de que consta el flujo de diseño, argumentando la selección de unas en detrimento de otras.

## 2.2. Protocolos del modelo OSI

Una decisión fundamental a abordar para este trabajo es sobre qué capas del modelo OSI [39] se centrarían las comunicaciones entre un *host* de control conectado a la nube y el nodo *Fog* en cuestión. Este trabajo ha centrado la atención en las dos capas principales en cuanto a envío y seguridad de la información se refiere: las capas de red y transporte. Se han estudiado diferentes alternativas de protocolos existentes y se ha enfocado hacia una arquitectura destinada a Internet de las Cosas (Figura 12).

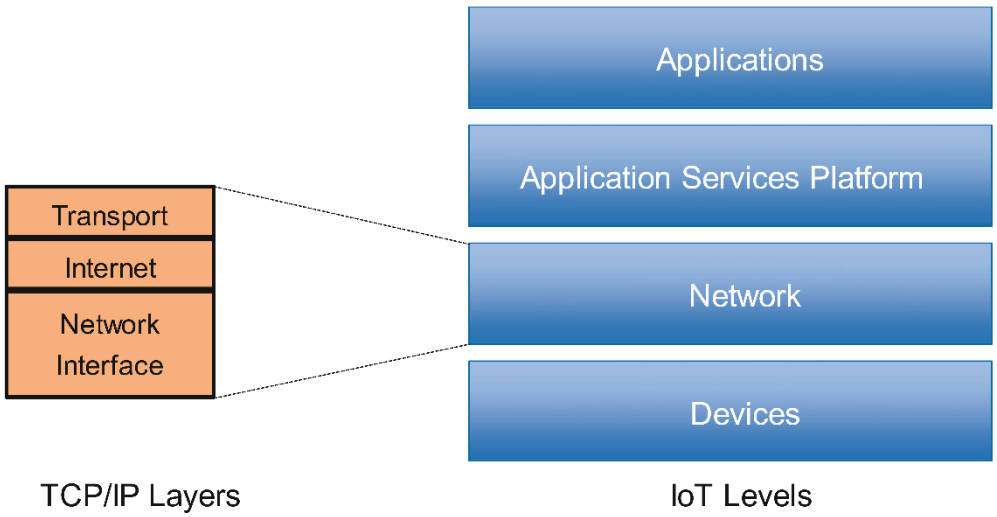



**Figura 12. Equivalencia de las capas OSI con los niveles de una arquitectura IoT [39].**

Centrándonos a continuación en el nivel de red, para una aplicación que requiere una conexión permanente a la red para el intercambio de información como es el Internet de las Cosas se hace imprescindible el empleo del protocolo IP. La versión IPv4, estandarizada en el año 1981, permite generar un total de 4.300.000.000 direcciones IP, que se estima que han sido cubiertas en su totalidad desde febrero del 2011 [40].

Con el surgimiento del IoT y otros nuevos servicios que requieren conexión a Internet, se espera que el número de dispositivos conectados a la red en el año 2020 sea cercano a los 50 mil millones. Ello supone un nuevo desafío al que hacer frente, al no contar con direcciones IP públicas disponibles con IPv4 [13][17].

Como respuesta a esta necesidad existente, surge a finales de los años 90 la versión 6 del protocolo, cuyos campos de direcciones de origen y destino de los paquetes poseen una longitud de 128 bits (frente a los 32 empleados en IPv4), lo que da lugar a $2^{128} = 3.4 \cdot 10^{38}$ direcciones IP públicas posibles. Como características adicionales se han incluido un conjunto de campos que mejoran la seguridad de las comunicaciones, así como la movilidad de los dispositivos que emplean estas direcciones. Ello favorece que los nuevos servicios conectados a la red como IoT estén orientados a comunicaciones, por lo general, basadas en protocolos inalámbricos de bajo consumo, como ZigBee o Bluetooth LE.

El principal problema que nos encontramos en este sentido es el hecho de que ambas versiones del protocolo IP no presentan interoperabilidad (IPv6 presenta ciertos modos de

transmisión para enviar información en formato IPv4), por lo que el proceso de migración de una versión a otra es ciertamente complejo [39].

En la Figura 13 se puede observar una comparativa entre las cabeceras de ambas versiones del protocolo, destacando especialmente el tamaño de los campos de direcciones y la simplicidad del *header* IPv6 frente al IPv4, que prescinde de una serie de campos que no se consideraron necesarios durante su estandarización. Algunos campos se han mantenido pero su designación ha sido modificada. Igualmente se han incluido nuevos campos, los cuales se citan a continuación:

a. El campo *Type of Service* de IPv4, que indica el tipo de información transmitida para determinar el tratamiento que le debe ser dado, se convierte en IPv6 en el campo *Traffic Class*.
b. Igualmente, *Time to Live* pasa a ser *Hop Limit* y señala el número máximo de saltos que puede dar por la red el paquete antes de ser descartado, para evitar que divague eternamente en bucle sin llegar a su destino.
c. *Protocol*, que se transforma en *Next Header* e indica el protocolo de la siguiente capa (la de transporte) que se empleará en la comunicación.
d. Asimismo, se ha añadido el campo *Flow Label*, que identifica a un flujo de datos que deben ser tratados de la misma forma (orientado a aplicaciones basadas en *streaming*).

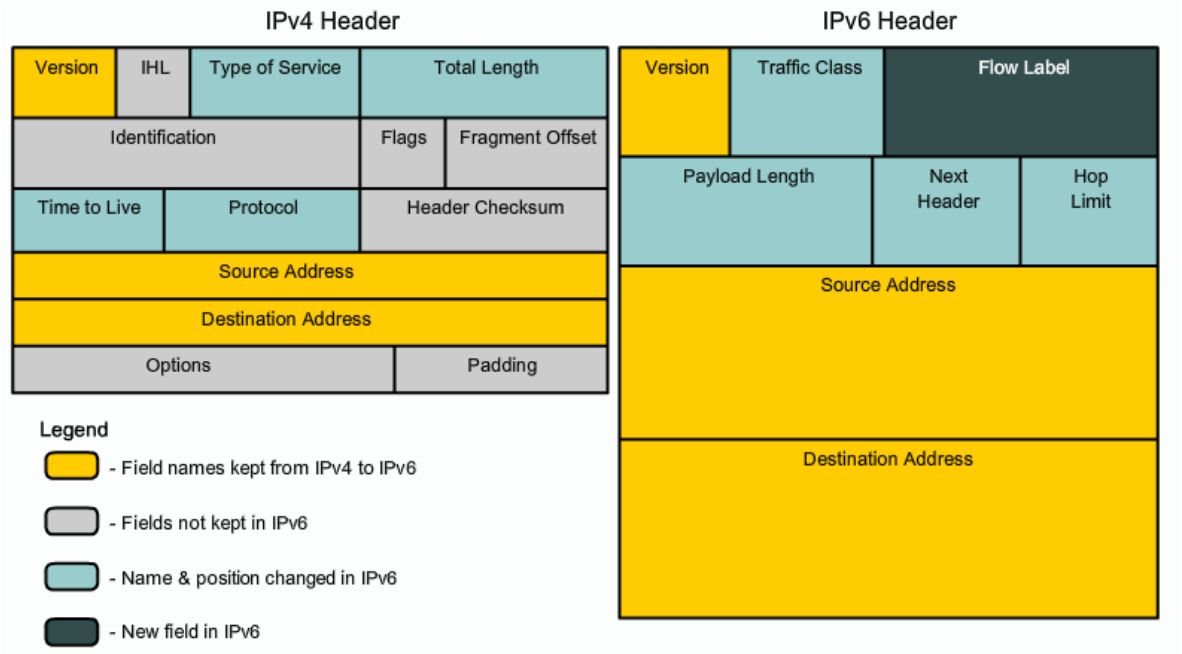



**Figura 13. Comparación entre las cabeceras de las versiones IPv4 e IPv6 [41].**

En cuanto a protocolos de transporte se refiere, el más ampliamente empleado en Internet es el protocolo TCP, constituyendo el modelo TCP/IP en el que se basan la mayoría de comunicaciones en la red. Sin embargo, para una arquitectura como IoT, los mecanismos de control de flujo empleados son demasiado complejos e incluso innecesarios. En la mayoría de los casos, es preferible el descarte de un paquete que los tiempos de espera por un determinado retardo en las comunicaciones, especialmente para aplicaciones basadas en tiempo real como la que nos concierne.

Por esta razón, se considera óptimo para este tipo de aplicaciones el empleo de UDP, protocolo no orientado a conexión y que no precisa de técnicas de establecimiento de sesión tipo *handshaking* o de mecanismos de control de flujo. Dichos mecanismos de control se realizan en capas superiores cuando sea requerido, como por ejemplo en la capa de aplicación [42]. Para verificar la integridad de los datos sí se incluye un campo de *checksum* [43], lo cual permite una integración idónea con IPv6. Si bien en IPv6 sobre TCP el campo de *checksum* no es obligatorio, sí que lo es cuando se utiliza UDP. Aparte de ello mantiene las direcciones y puertos entre origen y destino, donde algunos de estos puertos están predefinidos para determinados servicios (por ejemplo, el puerto 22 para conexiones SSH y el 80 para HTTP) y otros muchos libres para su uso por aplicaciones propias de los usuarios.

En la Figura 14 se muestra la cabecera equivalente empleando IPv6 como protocolo de red y UDP como protocolo a nivel de transporte. Como se puede apreciar, además de incluir las direcciones IP y los puertos origen y destino, se incluye la longitud del datagrama UDP y del *payload*, el campo de *checksum* para la verificación de los datos, el *Next Header* con valor 17 para indicar que el protocolo de transporte empleado es UDP, y finalmente un campo de ceros que hace un *padding* para alinear la palabra a una cantidad de *bytes* múltiplo de 32.

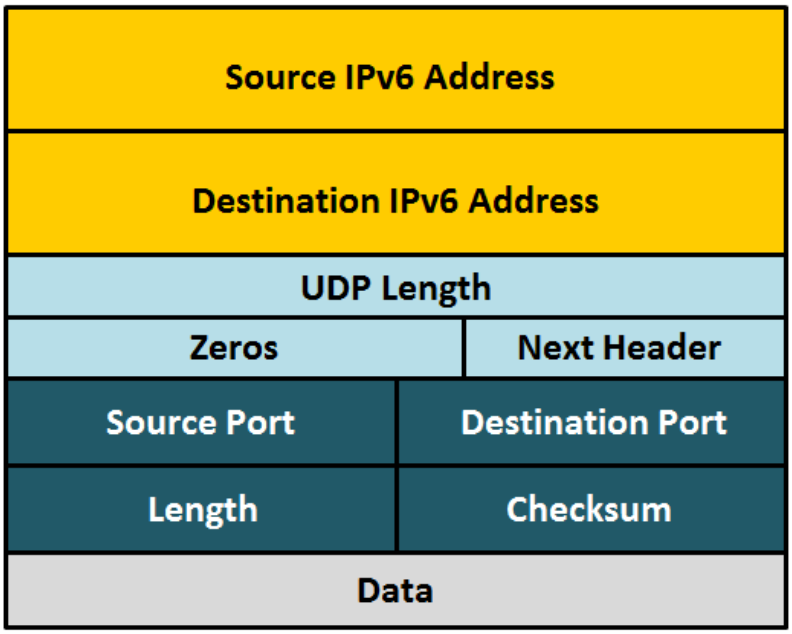



**Figura 14. Cabecera de IPv6 con UDP.**

## 2.3. Aplicaciones de seguridad analizadas

El abanico de alternativas seguras a implementar sobre un MPSoC programable como el que compondrá el *Fog Node* a desarrollar es bastante amplio, existiendo numerosas propuestas recogidas en la literatura especializada. Nos hemos centrado en el estudio, por una parte, de soluciones basadas en sistemas operativos empotrados (concretamente, en distribuciones Linux de este tipo), otras tipo *standalone*, pero con un enfoque *software* y, finalmente, en evaluar diferentes alternativas criptográficas que convivan con la aplicación, reduciendo así la complejidad de la solución final.

### 2.3.1. Linux empotrado

En primer lugar, surge la posibilidad de incorporar en alguno de los dos núcleos ARM de los que se dispone en el SoC un sistema operativo con *kernel* Linux, que permita establecer comunicaciones seguras mediante protocolos del tipo SSH entre el *Fog Node* y cualquier host conectado a la red. En este sentido, se optó por estudiar el rendimiento de dos distribuciones Linux distintas: Ubuntu y PetaLinux. El procedimiento de configuración de ambas distribuciones consiste en cargar el SO desde una tarjeta SD externa debidamente formateada tanto con los archivos de la distribución como con los necesarios para que dicha tarjeta fuera reconocida por el dispositivo Zynq.

La versión instalada de Ubuntu ha sido la 16.04 LTS (*Long Time Support*) la cual, a pesar de que cumplía con las condiciones de seguridad necesarias y permitía el establecimiento de sesiones SSH entre un equipo cualquiera de la red (autorizado por el sistema) y el dispositivo Zynq, presentaba inestabilidades significativas que producían fallos graves del tipo *kernel panic* cada cierto tiempo que forzaba la corrupción del sistema, teniendo que formatear la tarjeta SD y volver a configurar el sistema por completo (nueva instalación de los paquetes necesarios y configuración tanto de las interfaces de la red como del servidor SSH). Estos problemas con la distribución son graves, por lo que ha llevado a Xilinx a no reconocer Ubuntu como una distribución Linux soportada por sus dispositivos debido a su inmadurez tecnológica.

Por otro lado, se cuenta con la distribución PetaLinux [44], especialmente diseñada para ser incluida en dispositivos tipo SoC de Xilinx. En este caso, a pesar de que no se sufren los problemas de inestabilidad que se presentan con Ubuntu y de que el *kernel* cumple con la funcionalidad de seguridad deseada, se desestimó su utilización debido a que añade una complejidad que se ha visto innecesaria para este proyecto y que conduce a una degradación considerable de las prestaciones del sistema.

### 2.3.2. IPSec

Por otra parte, se ha analizado el *framework* de código abierto IPSec (*Internet Protocol Security*), que permite introducir funcionalidad a nivel de la capa de red tal como anti-repetición, confidencialidad, integridad y autenticación de la información en las comunicaciones mediante operaciones de cifrado entre dos equipos concretos que implementen esta política segura, determinando en cada momento el grado de seguridad (parámetro SPI) requerido tanto para una comunicación aislada como para un flujo constante de datos. Este conjunto de protocolos son una parte obligatoria de IPv6, siendo su uso opcional (y recomendable) en IPv4.

Su arquitectura (Figura 15) se fundamenta en el empleo de los protocolos AH y ESP, que serán descritos a continuación, junto con lo que se conoce como asociación de seguridad, un conjunto de algoritmos (tanto de cifrado como de autenticación) y parámetros seguros (como claves públicas y privadas) que se emplean para cifrar una determinada comunicación, siendo necesario que cada equipo o dispositivo cuente con su propia

asociación de seguridad. Para el manejo de números de secuencia de 64 bits en lugar de los 32 estándar, se hace necesario incluir la extensión DOI; en caso de emplear el formato estándar, este bloque es prescindible.

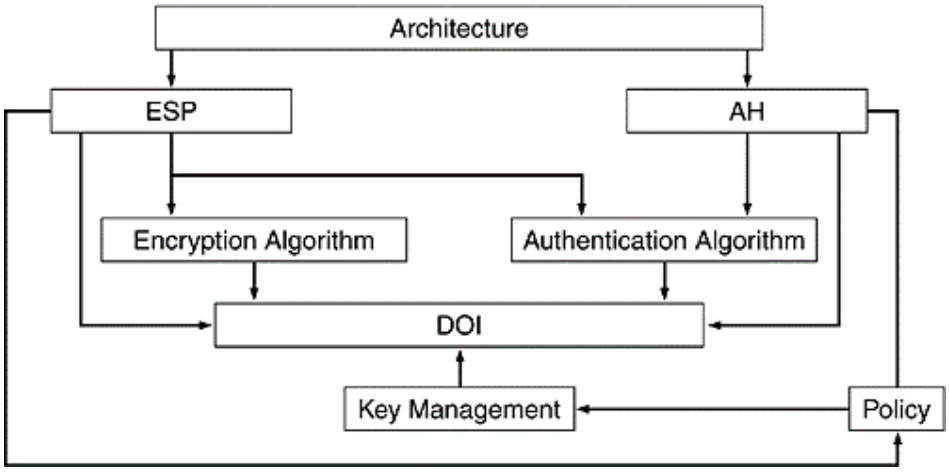



**Figura 15. Arquitectura IPSec. Adaptada de [45].**

El protocolo AH (Figura 16) proporciona autenticación e integridad a los paquetes enviados entre dos equipos, aplicando funciones *hash* (de forma general, SHA-256) mediante claves únicas que crean un resumen del mensaje. De esta forma, cualquier cambio realizado durante la transmisión será identificado por el receptor al aplicar exactamente la misma función *hash* que en el origen (necesidad de intercambio de claves inicial). Además, evita situaciones de repetición mediante la introducción de un bit adicional en la cabecera que permite indicar si el paquete ha sido recibido con anterioridad.

Una vez realizado el *hashing*, se genera una nueva cabecera IPSec que se incorpora al paquete original y que incluye, entre otra información, los parámetros SPI, la longitud del *payload* o el valor del *hashing* aplicado. Posteriormente, se transmite la información, siendo la nueva cabecera extraída en el destino y aplicando el *hash* correspondiente sobre el *payload* recibido, corroborando de esta forma que el valor resultante es el mismo que el calculado en el origen e incluido en la cabecera IPSec.

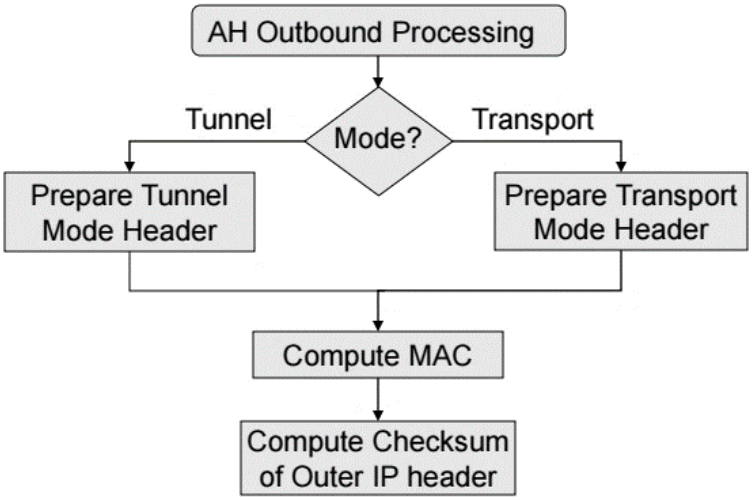



**Figura 16. Diagrama de flujo del protocolo AH de IPSec [46].**

El protocolo ESP (Figura 17) permite, además de la funcionalidad expuesta para AH (aunque su autenticación es más insegura), el cifrado de los datos enviados mediante la ejecución de algoritmos simétricos como pueden ser 3DES o AES, siendo posible elegir el grado de confidencialidad requerido en cada momento en función del tráfico actual. Lógicamente, en ambos extremos debe conocerse y emplearse el mismo algoritmo de cifrado para que el mensaje transmitido pueda ser reconstruido [47].

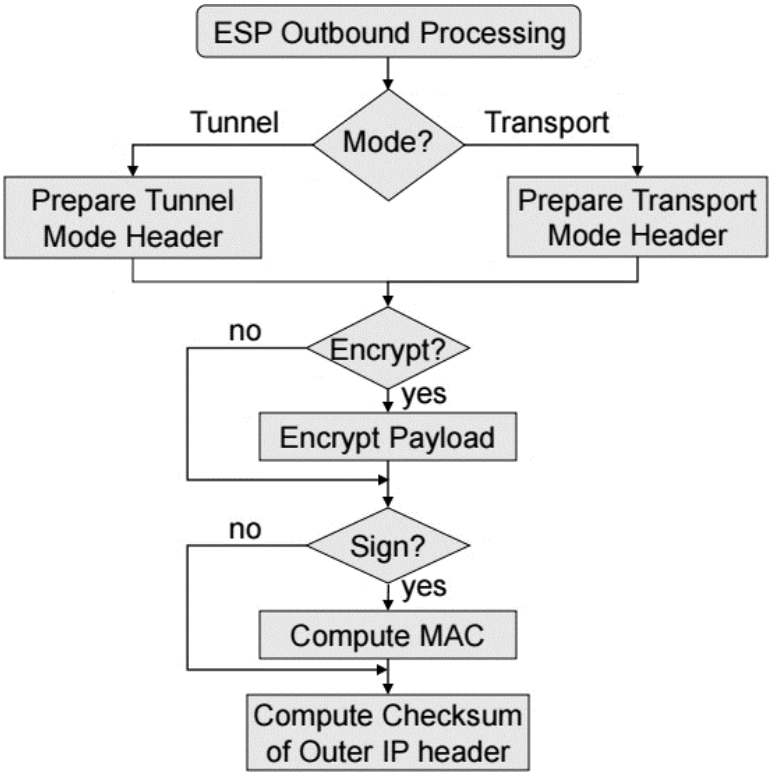



**Figura 17. Diagrama de flujo del protocolo ESP de IPSec [46].**

Tal como se puede observar en ambos diagramas de flujo, IPSec soporta dos modos distintos de funcionamiento, ya sea tanto empleando el protocolo AH como el ESP. En el modo túnel todo el paquete IP (cabecera y *payload*) es cifrado y autenticado, debiendo ser encapsulado dentro un nuevo paquete que permita el direccionamiento a través de la red. Esta alta protección hace que se trate de una solución empleada en servicios seguros en Internet como VPNs. Por su parte, el modo transporte únicamente se centra en cifrar el *payload* del paquete, por lo que no hay necesidad de cambiar el modo de direccionamiento, al permanecer la cabecera intacta. Por tanto, estamos ante una solución menos segura que en el caso del modo túnel, aunque puede ser suficiente para determinadas comunicaciones *host-to-host* [46].

En este sentido, en [48][49] se ha realizado una implementación de este *framework* sobre FPGAs con resultados satisfactorios en cuanto a frecuencia de trabajo se refiere. Sin embargo, el consumo de recursos en los dispositivos en los que han sido implementados es, por lo general, elevado para lo que es un único bloque IP, más teniendo en cuenta que la plataforma final constará de más módulos, además de los específicos destinados a seguridad. Dada la complejidad introducida por esta solución, se ha descartado su uso en este proyecto.

### 2.3.3. Alternativas criptográficas

Para evitar introducir complejidades innecesarias en la solución final, se ha estudiado la posibilidad de implementar en *hardware* aquella parte computacionalmente más costosa de alguna de las librerías criptográficas *software* más empleadas y reputadas de la actualidad, o bien hacer uso de bloques de cifrado compactos especialmente pensados para su implementación en *hardware*, para aplicaciones tales como Internet de las Cosas o sistemas de seguridad en entornos industriales y médicos.

#### *2.3.3.1. Librerías criptográficas*

En referencia a las librerías criptográficas, tras analizar el abanico de opciones disponibles, se ha optado por la librería criptográfica NaCl (*Networking and Cryptography library*), modelada en C/C++ por el grupo de trabajo de Daniel J. Bernstein, creador de las curvas elípticas Curve25519 [50].

El objetivo principal de esta librería es asegurar las comunicaciones a través de la red de la forma más simple posible en cuanto a complejidad del código se refiere, proporcionado confidencialidad e integridad robusta de los datos, así como seguridad adicional frente a ataques por *sniffing* o modificación del contenido de los paquetes Ethernet. Además, no permite configuraciones de baja seguridad como sí hacen otras librerías criptográficas, incluyendo siempre encriptación de los datos junto con el proceso de autenticación de las partes implicadas en el proceso de comunicación. Cabe pensar que utilizar configuraciones de alta seguridad conlleva un tiempo de ejecución excesivamente alto de la librería al tener que realizar mayor número de operaciones. Sin embargo, tras llevar a cabo comparaciones con librerías de ampliamente empleadas como OpenSSL [51], se obtienen mayores frecuencias de trabajo independientemente del tamaño del paquete de red y del modo de actuación (autenticación o encriptación) sobre los mismos dispositivos. Asimismo, el reducido número de líneas de código que incluye en comparación con otras librerías como la mencionada OpenSSL la hacen idónea para ser planteada como una opción a implementar en un sistema *hardware* empotrado, donde tanto la memoria, como la potencia disipada y los recursos lógicos disponibles son limitados.

La librería se compone de dos funciones básicas para cifrado y descifrado de información, `crypto_box` y `crypto_box_open`. Ambas tareas hacen uso de los mismos argumentos: un parámetro `m` que representa el mensaje a tratar, una variable `n` de 24 bytes denominada `nonce`, que se emplea conjuntamente con el mensaje `m` para generar el mensaje cifrado `c`; y las claves tanto públicas como secretas `pk` y `sk`. Para generar las claves públicas, cada ente del proceso de comunicación debe ejecutar previamente la orden `crypto_box_keypair`, que crea `pk` a partir de la `sk` propia de cada usuario [52].

A partir de este punto, se pueden añadir a la aplicación modelada funcionalidad adicional como, por ejemplo, intercambio de claves empleando Diffie-Helman o curvas elípticas, y autenticación mediante HMAC o Poly1305. Para el cifrado de la información se emplea por defecto el algoritmo Xsalsa20, algo más simple que AES (disponible también como opción a emplear) y más rápido en tiempo de ejecución.

El principal problema que presenta esta librería es el uso de lenguaje ensamblador en algunas partes de su implementación, lo que convierte en un proceso complejo su implementación en un dispositivo lógico como es el caso de una FPGA. Estas partes

optimizadas para su ejecución en un procesador de arquitectura x86 deberán ser reescritas para su migración a *hardware*.

Existen versiones alternativas derivadas de NaCl como es el caso de LibSodium. Está disponible en varios lenguajes de programación como C/C++ o Python y prescinde del lenguaje ensamblador en su modelado, además de estar disponible para múltiples arquitecturas de procesadores ARM, Intel o AMD y para una gran variedad de sistemas operativos tanto a nivel de *host* como Windows, Linux o MacOS X, donde es fácilmente instalable, como para sistemas operativos móviles como Android o empotrados como FreeRTOS o Contiki OS [53]. Sin embargo, el desmesurado uso de funciones estándar de C tanto para manejo de *strings* como para uso dinámico de memoria requieren, o bien de la instalación de PetaLinux para su gestión, o de una importante tarea de reestructuración de las funciones de las librerías.

Una última alternativa a tener en cuenta es TweetNaCl, una versión compacta de NaCl que hereda su nombre del tamaño que presenta, al ser posible incluir todo su código fuente en 100 *tweets* de 140 caracteres. Esta librería, disponible tanto en C como en Python, incluye las 25 funciones principales de la librería NaCl en un único fichero fuente, junto con su correspondiente fichero de parámetros e *includes*, evitando el largo árbol de directorios en los que se albergan los ficheros fuente tanto en NaCl como en LibSodium. Su simplicidad la hace propicia para una implementación *hardware* como la que tiene como objetivo el presente trabajo, alcanzando las mismas características de seguridad y mejorando las prestaciones en cuanto a velocidad se refiere de su librería original [54].

#### *2.3.3.2. Bloques de cifrado compactos*

Partiendo de la teoría criptográfica tradicional, un *block cipher* es un módulo de cifrado de clave simétrica (misma clave secreta para cifrar y descifrar la información) que trabaja sobre palabras de bits de longitud fija, aplicando funciones reversibles; esto quiere decir que, usando la misma clave, en el origen y el destino pueden obtener la información original únicamente empleando transformaciones opuestas.

En este ámbito, la referencia obligada es AES, algoritmo de cifrado estándar empleado en la mayoría de aplicaciones en red con el objetivo de encriptar las comunicaciones entre un origen y un destino. Es de suponer que, en un primer momento,

este tipo de algoritmos estaban pensados para dispositivos tipo *host* o servidores de cómputo, pero, con el surgir de tecnologías como el Internet de las Cosas, en la que se precisa de dispositivos finales lo más simples posibles, se requiere una renovación de las unidades de cifrado empleadas hasta ahora, en las que no se tenía en cuenta la cantidad de recursos y de memoria consumidos o el consumo energético derivado de su operación.

Siguiendo esta línea de actuación, surgen lo que se conoce como bloques de cifrado compactos, algoritmos criptográficos especialmente pensados para su implementación en sistemas *hardware* empotrados con recursos lógicos limitados y un reducido consumo de energía, sin comprometer el nivel de seguridad de la información cifrada [55]. En cuanto a tipos de *block ciphers*, podemos distinguir entre dos grandes grupos:

- ***Substitution-Permutation Network* (SPN).** Utiliza los conceptos básicos de la teoría de la información de Shannon y es el principio básico de funcionamiento del AES. Se basa en la ejecución de dos tareas distintas. En primer lugar, se realiza lo que se denomina **confusión**, que no es más que hacer lo más complicada posible la relación entre los bits de entrada, la clave secreta y el texto cifrado final. Esta etapa hace uso de lo que se denominan *S-boxes* o *substitution boxes*, que actúan como *Lookup Tables*, por lo general, dinámicas (cambian con la clave única empleada) que establecen una relación, lineal o no, entre los bits aplicados a la entrada y los obtenidos a la salida.

  Por otro lado, se aplica la **difusión**, que se centra en lograr un efecto avalancha o, dicho de otro modo que, si se modifica una parte del texto plano aplicado como entrada, dicho cambio debe verse reflejado en el texto cifrado modificando la mayor cantidad de bits posible en el texto cifrado. Para ello se emplean las *P-boxes* o *permutation boxes*, que realizan operaciones de permuta, desplazamiento o rotación en todo el espacio resultante [56]. Cabe destacar como bloques de cifrado compactos orientados a *hardware* de este tipo el PRESENT, el PRINCE y el mCrypton. En la Figura 18 se muestra el diagrama de bloques de este tipo de cifrado:

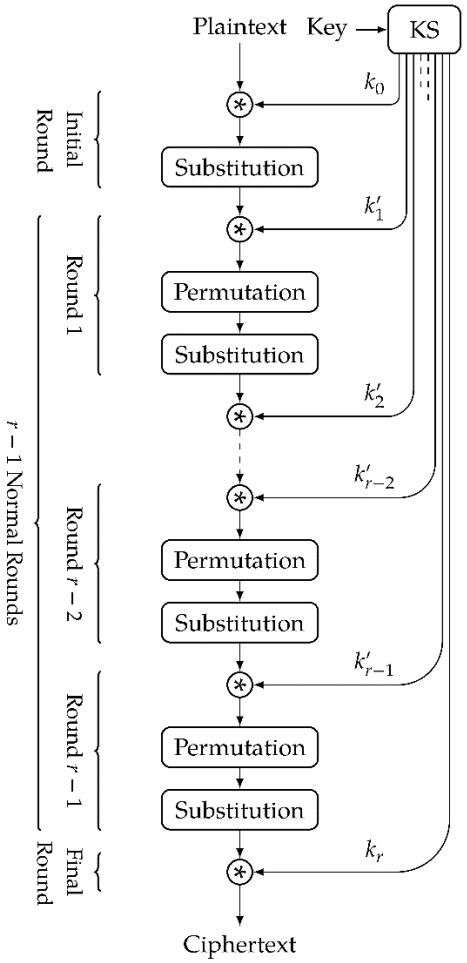



**Figura 18. Diagrama de bloques del cifrado SPN [56].**

- **Redes de Feistel.** Esta metodología de cifrado se basa en rondas, en las cuales se realizan siempre las mismas operaciones, obteniéndose, de forma general, un algoritmo más robusto mientras más rondas se realizan [56]. Su funcionamiento básico se basa en dividir la palabra de entrada en dos fragmentos de igual longitud y se realizan operaciones (binarias, aritméticas, etc.) entre sólo una de ellas y la clave única empleada, trabajando con el otro fragmento en la siguiente ronda (Figura 19). En este grupo, cabe destacar los bloques de cifrado HIGHT, Simon y Speck.

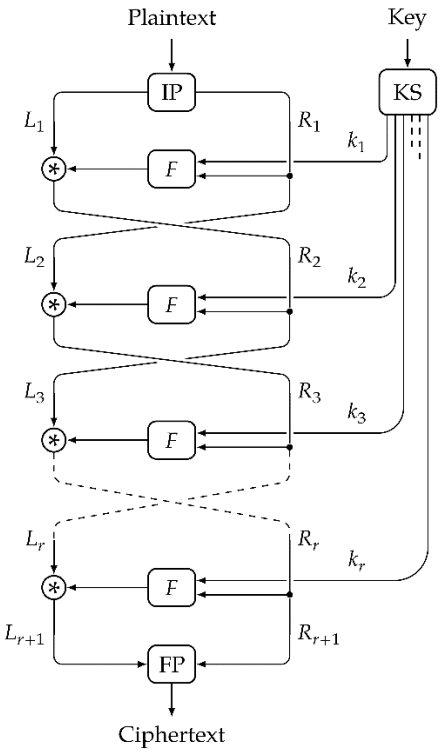



**Figura 19. Diagrama de bloques del cifrado basado en redes de Feistel [56].**

Tras analizar diversos estudios disponibles en el estado del arte sobre la implementación de este tipo de módulos en *hardware* [57]–[59], hemos seleccionado para este trabajo el bloque de cifrado Simon. Este módulo, junto con el bloque Speck (más orientado a implementaciones *software*), fueron desarrollados por la NSA en el año 2013 con el objetivo de obtener sistemas de cifrado para aplicaciones de Internet de las Cosas flexibles y eficientes en términos de consumo de recursos y de energía, sin descuidar en ningún momento la robustez en cuanto a la seguridad del cifrado de la información se refiere [60].

Tal como se ha mencionado anteriormente, se trata de un algoritmo de cifrado basado en redes de Feistel realizando, en cada ronda, operaciones binarias tales como XOR, AND y desplazamientos a la izquierda (Figura 20). Además, posee distintos modos de configuración, manejando datos de entrada en formatos de 32, 48, 64, 96 y 128 bits, a partir de los cuales se determina el tamaño que debe tener la clave única, así como el número de rondas a realizar para garantizar un sistema lo suficientemente seguro [61].

Por estas características, se le considera una opción idónea y viable para una implementación *hardware* como la considerada en el presente trabajo, por lo que se ha

optado por realizar su modelado y análisis (ver en el capítulo correspondiente al Diseño de la etapa de cifrado del sistema).

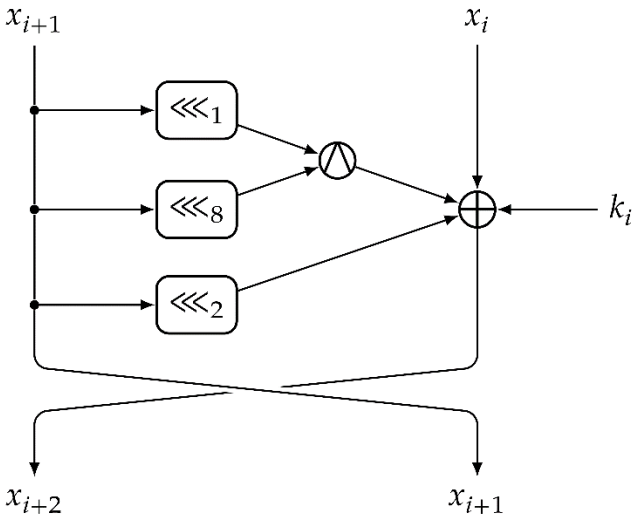



**Figura 20. Diagrama de bloques del cifrado Simon [56].**

## 2.4. Síntesis de Alto Nivel

El imparable crecimiento que ha sufrido en los últimos años la capacidad de integración en los chips, que ha dado lugar a la integración en un mismo encapsulado de gran número de componentes heterogéneos (tanto *hardware* como su *software* asociado), así como la necesidad de reducir el *time-to-market* para situar un producto en el mercado con una clara ventaja competitiva, han provocado que el modelado de un sistema no pueda basarse por completo en una descripción RTL, la cual presenta largos tiempos de desarrollo a medida que la complejidad del sistema aumenta. Por tanto, se hace necesario abstraer el modelado de la funcionalidad deseada a niveles más altos, permitiendo acortar los plazos de diseño y reducir los costes asociados al desarrollo de un producto [62].

En este punto, entra en juego la síntesis de alto nivel (HLS por sus siglas en inglés), que parte de un modelo en lenguaje de alto nivel, ya sea con información temporal (usando por ejemplo SystemC) o sin ella (generalmente en C/C++), para posteriormente realizar una transformación de la funcionalidad modelada a su microarquitectura, formada generalmente por una ruta de datos y su unidad de control representada a nivel RTL, la cual debe ser equivalente y tener en cuenta las restricciones de diseño definidas en el nivel de abstracción superior, con el fin de cumplir los objetivos de latencia, área y consumo de potencia.

De forma resumida, lo que se hace durante este proceso es explorar posibles soluciones arquitecturales dentro del espacio de diseño a partir de la funcionalidad definida y de las restricciones impuestas, siendo seleccionada de forma automática por la herramienta de síntesis de alto nivel aquella que más se ajusta a los criterios de diseño. De este modo, se consigue aumentar la productividad de la labor del diseñador *hardware* y reducir los tiempos de desarrollo [63].

En la Figura 21 se muestra el flujo de diseño detallado del proceso de síntesis de alto nivel, partiendo de una especificación modelada con un lenguaje de alto nivel de abstracción y hasta llegar a la obtención de la arquitectura a nivel RTL.

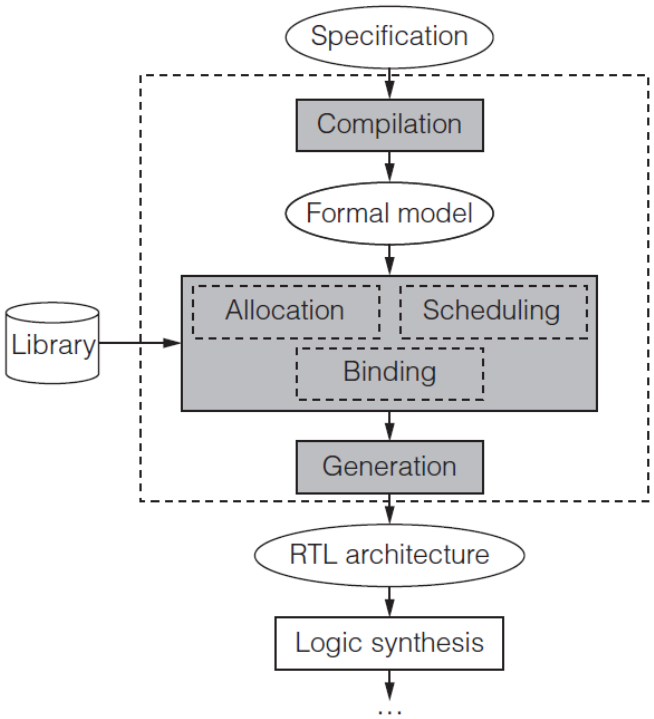



**Figura 21. Flujo de diseño de la metodología de síntesis de alto nivel [64].**

En este sentido, es importante resaltar el papel que desempeñan las FPGAs dentro de esta metodología. Como primera característica, cabe destacar la rapidez en cuanto al prototipado que presenta este tipo de dispositivos permitiendo, en caso de no cumplir la funcionalidad modelada con lo descrito en las especificaciones iniciales, una total flexibilidad para implementar las modificaciones funcionales requeridas sin incurrir en costes de fabricación adicionales frente a los costes NRE que supone la fabricación de un ASIC.

Por otro lado, hay que resaltar su capacidad para el desarrollo de sistemas basados en plataformas ya que, de forma general, la mayoría de FPGAs actuales incluyen un conjunto

de bloques IP estándar (así como la implementación de los buses de comunicación) que permiten reducir los plazos de desarrollo, favoreciendo el *time-to-market*, y aumentar la calidad de los resultados obtenidos.

Finalmente, en relación a la tendencia actual que existe respecto a la computación de altas prestaciones (HPC - *High-Performance Computing*) o procesamiento masivo de datos para aplicaciones tales como análisis financiero, Big Data o procesamiento multimedia, las FPGAs o los SoCs que incluyen lógica programable constituyen un método eficaz de aceleración, cuya funcionalidad es menos compleja de implementar con un alto nivel de abstracción que si se realizara empleando lenguajes HDL, lo que dispararía de forma exponencial los tiempos de diseño [65].

## 2.5. Lenguajes de modelado

En cuanto al modelado en alto nivel, hay que resaltar inicialmente que el catálogo de alternativas de que dispone el diseñador ya no se limita sólo al lenguaje C y algunas de sus derivaciones y evoluciones (C++ o SystemC) sino que, en la actualidad, es posible incluso diseñar aplicaciones en Python que sean integradas en los núcleos procesadores del SoC y que no precisen de conocimientos arquitecturales de cómo se programa el *hardware* por parte de los ingenieros y desarrolladores [66].

Sin embargo, en este caso, se ha planteado únicamente como opciones el uso de los lenguajes C/C++ y SystemC, ya que son más eficientes si se tienen adquiridos los conocimientos y las aptitudes para el diseño *hardware* y, por lo general, permiten obtener mejores resultados en términos de prestaciones, área y consumo de potencia. A continuación, se realizará un breve análisis sobre las principales características de ambas alternativas y se justificará la elección tomada para la realización de este trabajo.

La principal ventaja que presenta emplear C/C++ radica en el hecho de que ofrece un nivel de abstracción alto, no siendo necesario (ni posible) tener en cuenta ningún tipo de restricción temporal a la hora de desarrollar la funcionalidad deseada, lo cual facilita bastante la partición *hardware/software* inicial del sistema. La única forma de guiar a la herramienta de síntesis es mediante la aplicación de ciertas directivas que permitan

optimizar algún tipo de parámetro del sistema, ejecutándose el código de forma secuencial salvo que se le indique aplicar paralelismo a bloques funcionales que lo soporten.

Como desventajas cabe destacar que no es posible controlar de forma minuciosa las interfaces de comunicación o las señales internas del bloque IP a generar, proceso que se realiza de forma automática y transparente al diseñador; esto puede provocar una cierta discrepancia entre el modelo C/C++ definido y el código RTL generado por la herramienta de síntesis de alto nivel durante la etapa de cosimulación, siendo necesarias modificaciones adicionales para ajustar sus comportamientos, lo cual no es sencillo, ya que sólo es posible por parte del diseñador modificar un algoritmo que en lenguaje de alto nivel funciona perfectamente (y que puede dejar de hacerlo al aplicar cambios) o alguna de las estrategias de optimización definidas.

Por otra parte, SystemC es una librería de C++ que sí permite modelar *hardware* teniendo en cuenta tanto aspectos de funcionalidad como temporales y arquitecturales, característica principal que le distingue con respecto a C/C++. El diseñador puede definir una planificación temporal precisa que guíe al proceso de síntesis de alto nivel, obteniendo una descripción RTL totalmente a medida de las necesidades definidas.

De esta forma, se pueden establecer qué operaciones se realizan en cada ciclo de reloj o sincronizar procesos de forma precisa, favoreciendo el paralelismo de las operaciones y, por tanto, aumentando las prestaciones del sistema; en este caso, resulta necesario tener unos conocimientos básicos sobre la tecnología de implementación empleada con el fin de poder definir qué funcionalidad es capaz de realizar el sistema en un ciclo de reloj ya que, si se sobrepasa la capacidad del sistema, la frecuencia de funcionamiento se verá afectada, sufriendo un descenso. En resumen, SystemC se trata de la elección adecuada cuando se procede a modelar un sistema cuya funcionalidad debe estar ceñida a una planificación temporal estricta, siendo necesario controlar la señalización interna del bloque IP y sus interfaces de entrada/salida. SystemC también soporta otros estilos de modelado, como por ejemplo TLM 2.0 [67].

Para el modelado de los bloques IP de que constará el diseño planteado en este trabajo, se ha decidido emplear C/C++, ya que el sistema requerido no requiere un control temporal estricto al ser la funcionalidad implementada puramente algorítmica; de esta forma, en etapas tempranas del flujo de diseño, se prescinde de limitaciones por parte del

diseñador en cuanto a latencia, planificación de las tareas y la utilización de los recursos disponibles en la lógica programable, ya que son impuestas directamente por las herramientas de síntesis de alto nivel, las cuales se encargan, a su vez, de organizar y sincronizar los distintos procesos de que consta la funcionalidad implementada para intentar alcanzar una estructura orientada a *dataflow.* Finalmente, hay que indicar que un último aspecto determinante a la hora de emplear este lenguaje en detrimento de SystemC ha sido poder reusar parte de código diseñado con anterioridad [68], proceso que está adquiriendo cada vez mayor importancia en el ámbito de la ingeniería y que permite reducir considerablemente los tiempos de diseño[69].

## 2.6. Entorno de diseño

A continuación, se explican las diferentes herramientas que conforman el flujo de diseño utilizado. Se incluyen en este flujo herramientas de síntesis, tanto de alto nivel como de síntesis lógica, de verificación y de implementación de la plataforma.

### 2.6.1. Síntesis de alto nivel

En un flujo de diseño basado en síntesis de alto nivel, la herramienta de síntesis resulta crítica en cuanto a la calidad de los resultados finales obtenidos en términos de PPA (*Power, Performance, Area*). La generación a nivel RTL de arquitecturas eficientes a partir del modelo de alto nivel afectará de forma notable en etapas posteriores del flujo de diseño, como son la síntesis lógica y la implementación. En cualquier caso, la arquitectura obtenida debe ajustarse a las especificaciones iniciales aportadas por la aplicación a la que será destinado el acelerador.

Para el presente TFM, se han estudiado dos herramientas de diseño de alto nivel a emplear: Xilinx Vivado HLS y Cadence Stratus.

Vivado HLS (Figura 22) es la herramienta del entorno Vivado de Xilinx dedicada al diseño en alto nivel. Permite transformar un modelo algorítmico con un alto nivel de abstracción en su implementación *hardware*, pudiendo realizar el modelado en lenguajes tales como C/C++, SystemC u OpenCL. Asimismo, aporta un conjunto de directivas que permiten al diseñador controlar el proceso de síntesis de alto nivel, proporcionando a la

herramienta una serie de restricciones a tener en cuenta para realizar la búsqueda de la mejor solución en el espacio de diseño [33].

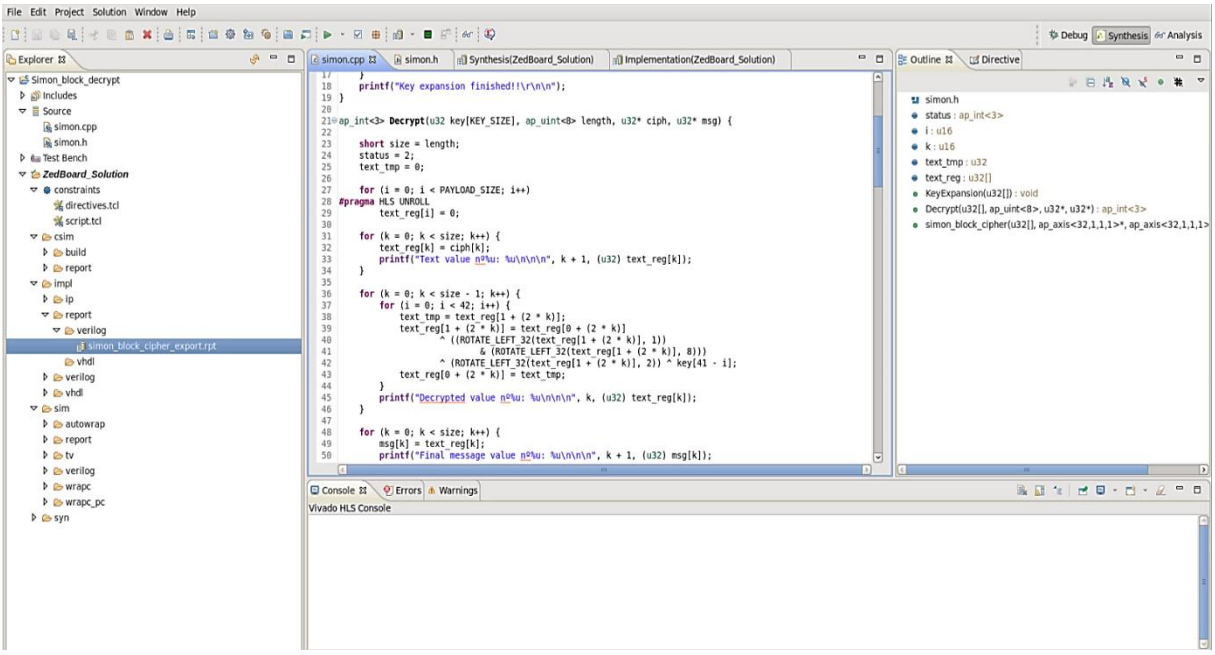



**Figura 22. Entorno de Vivado HLS.**

La definición y creación de las interfaces de comunicación en Vivado HLS se realiza a través de directivas específicas, que permiten simplificar este proceso infiriendo de forma automática el tipo de interfaz AMBA AXI deseado según el estilo de modelado de interfaz requerido. Al tratarse de un proceso muy automatizado, la herramienta generará las interfaces, garantizando el cumplimiento de los estándares en comunicación. En el caso de utilizar lenguajes de modelado que soporten especificaciones precisas a nivel de ciclo de bus (BCA – *Bus Cycle Accurate*), como es el caso de SystemC, permite el control de las señales internas y la planificación temporal de las operaciones realizadas por el bloque, en un proceso interactivo guiado por el diseñador en busca de alcanzar las mejores prestaciones posibles.

Por su parte, Cadence Stratus es el entorno de síntesis de alto nivel de Cadence que toma el testigo de CtoS (*C-to-Silicon*). Al igual que Vivado HLS, permite la generación de una descripción RTL a partir del modelado de la funcionalidad en alto nivel (Figura 23), por lo general en SystemC (para hacerlo en C/C++, es necesario incluir en un paso final un *wrapper* en SystemC). Igualmente soporta la verificación del diseño tanto en alto nivel, a nivel RTL o una cosimulación multinivel (incluyendo visualización de formas de onda y otras facilidades

incluidas en la metodología de verificación soportada), para lo que es necesario incluir un simulador externo, ya sea Cadence Incisive o Mentor ModelSim integrado con el flujo de síntesis. Asimismo, realiza de forma prácticamente automática una búsqueda y comparación de múltiples arquitecturas posibles para alcanzar los requisitos de prestaciones, área y consumo de potencia especificados previamente por el diseñador, además de proporcionar una serie de directivas concretas para que sean empleadas en aquellas zonas del programa que se consideren oportunas [70].

Como inconvenientes principales cabe destacar el hecho de que precisa de la importación de las librerías del dispositivo concreto que se va a emplear, tanto de los bloques principales como de los buses de comunicación, proceso que Vivado HLS realiza de forma nativa para los dispositivos de Xilinx, así como de un *makefile* en el que se definen todas las características del proyecto y que se encarga de guiar el proceso completo de verificación de la funcionalidad y la síntesis de alto nivel.

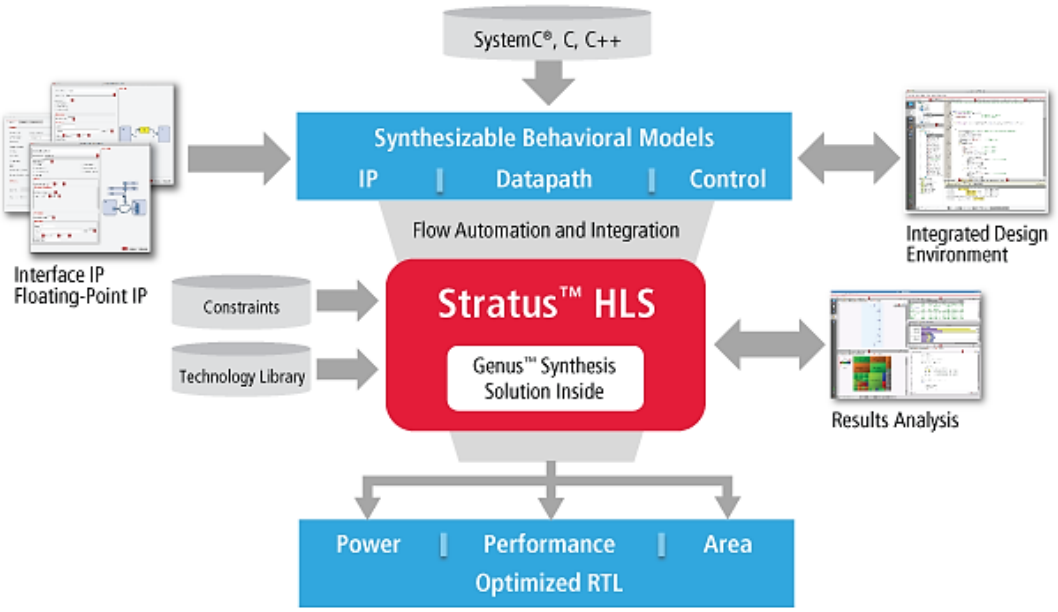



**Figura 23. Flujo de diseño de Cadence Stratus** [70]**.**

Una característica que se ha observado estudiando ambas herramientas de síntesis de alto nivel es el hecho de que Vivado HLS permite alcanzar frecuencias más altas sobre los dispositivos de Xilinx, ya que añade registros en aquellas rutas que sean críticas del diseño con el único fin de reducir la cantidad de operaciones a realizar en cada ciclo de reloj. Sin

embargo, en algunos casos esta característica puede convertirse en un inconveniente (por ejemplo, en interfaces CABA), ya que la introducción de nuevos registros puede conducir a una desincronización de las tareas a realizar, provocando un fallo de la funcionalidad del bloque IP.

### 2.6.2. Síntesis lógica e implementación

En el caso de la síntesis lógica, las opciones que se han evaluado para transformar la descripción RTL obtenida mediante síntesis de alto nivel hacia su implementación son Xilinx Vivado y Synopsys Synplify Premier Design Planner, independientemente de si se ha empleado Vivado HLS o Cadence Stratus como herramientas de síntesis de alto nivel.

Xilinx Vivado [32] se basa en el diseño de plataformas (conjunto de bloques IP) para generar el sistema final, lo que tiene como consecuencia directa que las prestaciones finales obtenidas tras la síntesis lógica están condicionadas por aquellas partes críticas del diseño y que puedan limitar las prestaciones de los bloques IP modelados en alto nivel.

A pesar de este inconveniente, la herramienta aplica directivas y estrategias de optimización especificadas por el diseñador tanto en la etapa de síntesis lógica como en la de implementación (*place & route*). Dichas restricciones tienen por objetivo reducir los retardos en las interconexiones entre las distintas partes de que consta el sistema o reducir la cantidad de recursos empleados para implementarlo sobre la lógica programable de la FPGA. En esencia, su objetivo es optimizar el tiempo de ciclo minimizando el uso de recursos.

Por su parte, Synplify [71] realiza la síntesis lógica completa del sistema con mayor eficiencia con respecto al mismo proceso realizado en Vivado, o de bloques determinados del diseño, obteniendo, de forma general, mayores frecuencias de trabajo. Sin embargo, las versiones evaluadas no tienen una integración completa con el entorno de Vivado, ya que presenta problemas con los *drivers software* de los bloques IP a la hora de desarrollar el *software* empotrado. Asimismo, incluye la posibilidad de aplicar estrategias de optimización propias a cada bloque y de analizar las rutas críticas del bloque IP para poder reducir los retardos una vez ya ha mapeado los recursos lógicos disponibles en la FPGA. Tras completar el proceso, se genera un archivo EDIF que contiene el *netlist* del bloque, pudiendo ser integrado en una plataforma de Vivado como una *Black Box*.

### 2.6.3. Diseño de la plataforma

Xilinx Vivado es la herramienta de referencia para la implementación final del sistema sobre una plataforma FPGA para la generación del *bitstream* final de la plataforma. Igualmente soporta la generación del *software* empotrado necesario para el sistema empotrado final.

Además, incluye utilidades para el depurado del sistema, su análisis de prestaciones y soporte para reconfigurabilidad.

### 2.6.4. Entorno integrado SDSoC

SDSoC es una herramienta desarrollada recientemente por Xilinx que pretende integrar en un mismo entorno la gestión de todas las etapas de que consta el flujo de diseño de un SoC programable, simplificando el proceso de transformación algorítmica hacia el *hardware*. En comparación con Vivado HLS, SDSoC permite realizar la partición *hardware/software*, utilizando como criterio su carga computacional y migrando a *hardware* aquellas partes más complejas, permaneciendo el resto de la aplicación en *software*, que será ejecutado en los procesadores que incluye el SoC. Para tomar esta decisión, se apoya en el uso de herramientas de perfilado *software* que se ejecutan directamente sobre los núcleos ARM de los SoCs programables de Xilinx.

Tomando como referencia esta partición, realiza la síntesis de alto nivel, la síntesis lógica y la implementación del sistema final, de forma automática y transparente para el diseñador el cual, si no indica restricciones concretas (lanzando los entornos de Vivado y Vivado HLS para aplicar las *constraints* y estrategias de optimización oportunas), deja en manos de la herramienta la toma de decisiones durante las distintas etapas de que consta el flujo de diseño. El modelo de uso de este entorno está enfocado hacia los desarrolladores de *software* que se encuentren realizando una primera aproximación al ámbito del *hardware* reconfigurable, con el objeto de acelerar su arquitectura.

Sin embargo, hay que resaltar que la experiencia con las herramientas de perfilado integradas no ha resultado demasiado satisfactoria, al no proporcionar excesiva información sobre la carga computacional de cada una de las funciones que conforman la aplicación. Otro aspecto a destacar es que cualquier tipo de ejecución o depurado de la funcionalidad se

vuelca siempre sobre los núcleos ARM del SoC empleado, no soportando la posibilidad de lanzarlo de forma simultánea sobre el *host* para obtener resultados puramente *software* de la arquitectura del sistema de referencia.

Otro aspecto es el soporte de la herramienta. Aunque se trata de una herramienta potente en cuanto a las características que integra, el hecho de que sea relativamente reciente conduce a que, en caso de encontrar alguna anomalía o problema en su manejo, resulte bastante difícil encontrar algún tipo de información útil tanto en la documentación oficial del entorno como en los foros de desarrolladores.

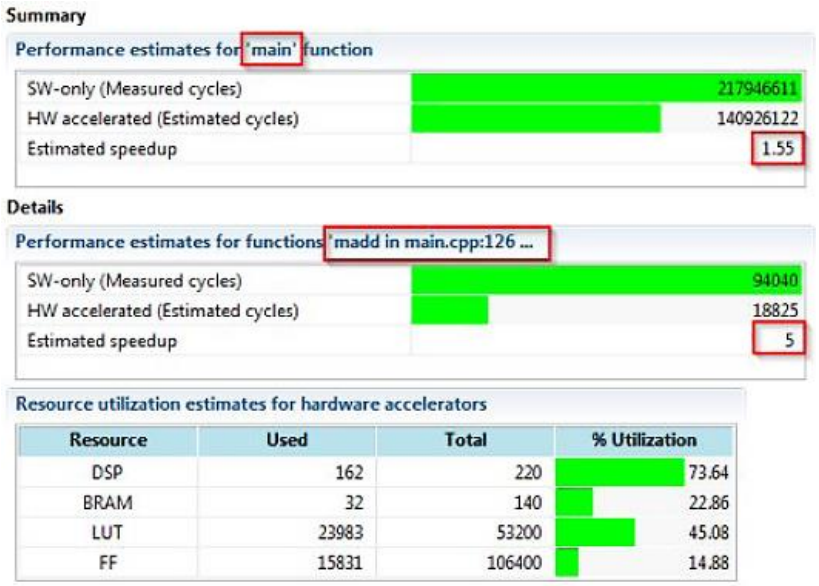



**Figura 24. Ejemplo de speedup y recursos en SDSoC [37].**

Finalmente, para el desarrollo de este TFM se ha optado por emplear como herramienta de síntesis de alto nivel Vivado HLS, debido a los buenos resultados proporcionados para modelos que no precisan de estrictas restricciones temporales y los resultados positivos alcanzados en proyectos anteriores en la división SICAD. Para la posterior síntesis lógica de la descripción RTL y su implementación sobre la FPGA se ha hecho uso de la herramienta Xilinx Vivado.

Adicionalmente, se ha decidido realizar también una aproximación al empleo de la herramienta SDSoC, con el objetivo de obtener una experiencia preliminar que permita determinar su utilidad y la calidad de los resultados obtenidos para poder considerarla como una alternativa en futuros trabajos de la división.

En la Figura 25 se resume el flujo de diseño seguido, indicando en cada etapa la herramienta empleada:

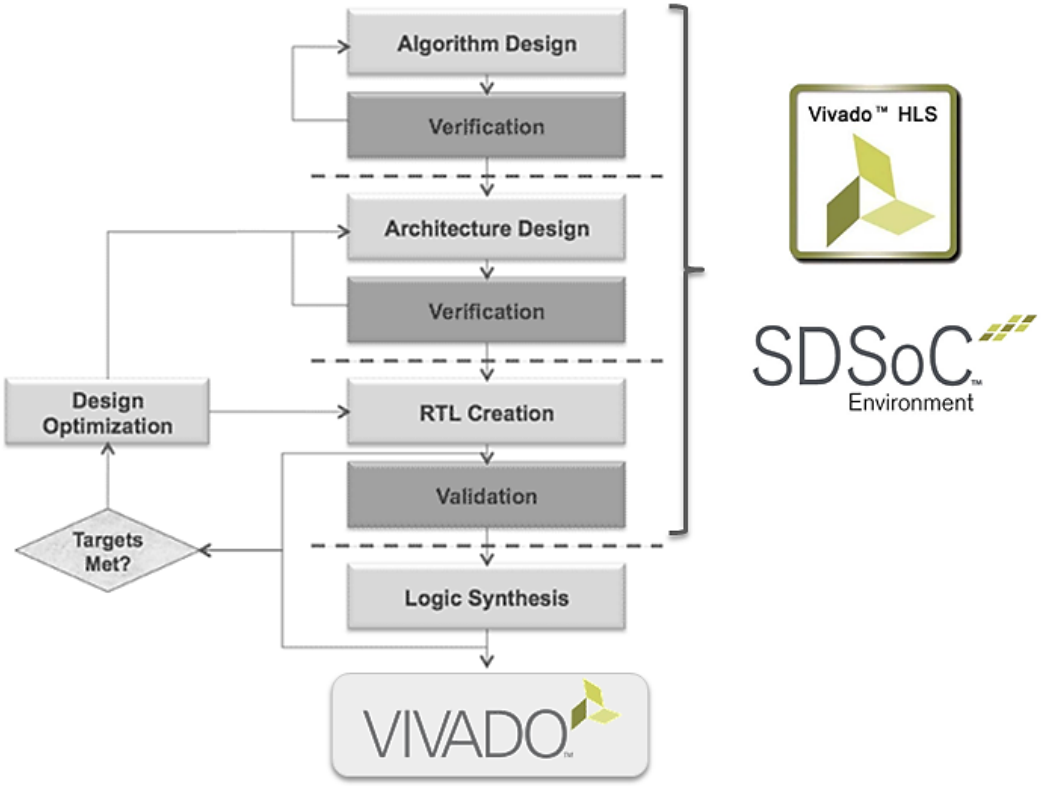



**Figura 25. Herramientas empleadas en cada etapa del flujo de diseño.**

## 2.7. Conclusión

Como alternativas a tener en cuenta para la etapa de cifrado del sistema, en el presente TFM se analizan los resultados derivados de la implementación de la librería criptográfica TweetNaCl empleando la herramienta SDSoC y se modela el bloque de cifrado Simon mediante el lenguaje C/C++ en Vivado HLS, comparando finalmente los resultados obtenidos en ambos casos y optando por aquella implementación más satisfactoria en términos de latencia y consumo de recursos lógicos.

Respecto a la etapa de autenticación, se ha resaltado la importancia de emplear como protocolos de comunicación IPv6 a nivel de la capa de red y UDP para la capa de transporte, para una aplicación vinculada a IoT como la que centra nuestra atención en este trabajo.

Por lo expuesto anteriormente, se ha optado por implementar los bloques IP que conformarán el sistema final del presente Trabajo Fin de Máster siguiendo una metodología

de síntesis de nivel y haciendo uso del lenguaje de modelado C/C++ debido, esencialmente, a que no precisamos de un control temporal exhaustivo de las señales internas de los bloques (funcionalidad puramente algorítmica) y la posibilidad de reutilizar en algunas etapas código anteriormente modelado.

En cuanto a las herramientas de diseño se ha optado, tal como se ha expuesto anteriormente, por emplear en la mayor parte del trabajo Vivado HLS, debido a que presenta buenos resultados para modelos en C/C++ sin restricciones temporales y permite guiar ciertas partes del proceso de síntesis mediante la utilización de las directivas oportunas, además de contar con experiencias previas en la realización de otros proyectos empleando dicha herramienta. Tanto para la síntesis lógica como para la implementación del sistema completo se ha decidido emplear Vivado, ya que se trata de un IDE completo que permite realizar ambos procesos de forma eficiente, proporcionando estrategias de optimización para aquellos parámetros del sistema que se consideren críticos y un entorno de visualización de resultados bastante intuitivo y simple para el diseñador.

# Capítulo 3. Diseño de la etapa de autenticación

## 3.1. Introducción

En este capítulo se aborda el diseño, empleando una metodología de síntesis de alto nivel, de los bloques IP que realizan las tareas de filtrado a nivel de las capas de red y transporte de los protocolos empleados en una arquitectura IoT típica. Además del modelado de la funcionalidad en sí, se realizan las etapas de verificación del código en lenguaje de alto nivel (C/C++), la síntesis de alto nivel para obtener la descripción RTL, la cosimulación para comprobar que los módulos se comportan en lenguaje HDL de la misma forma que en alto nivel y la exportación final de los bloques para su integración con el resto del sistema.

## 3.2. Diseño de los filtros de paquetes de red

### 3.2.1. Conceptos preliminares

De forma previa a la descripción detallada del flujo de diseño de los bloques IP, se hace necesaria la introducción de conceptos fundamentales para entender cómo funcionan los filtros en cuestión.

#### *3.2.1.1. Estructura Bloom Filter*

En primer lugar, se comenzará explicando el concepto de *Bloom Filter* (a partir de ahora, BF), estructura de datos en la que se basan los filtros diseñados.

Los BF permiten, mediante el uso de técnicas probabilísticas, determinar si un elemento concreto consultado pertenece a un conjunto específico. Con este método, se

consigue una reducción significativa del tiempo de análisis de la información, por lo que su empleo es frecuente en aplicaciones vinculadas al análisis de información en una red de datos. Asimismo, se trata de una estructura que demanda un consumo muy reducido de recursos lógicos, el cual dependerá, de forma directamente proporcional, del tiempo de respuesta que consideremos necesario para cumplir con las especificaciones de tiempo real de nuestro sistema y de la cantidad de elementos incluidos en el conjunto objeto de análisis. Finalmente, hay que resaltar que se trata de un mecanismo constante en cuanto al tiempo de respuesta se refiere, independientemente del tamaño de la información analizada.

Sin embargo, como cabría esperar, no todo son ventajas. El principal inconveniente que presentan estas estructuras es la posibilidad de que se devuelvan "falsos positivos"; dicho de otra forma, la respuesta proporcionada puede indicar que el elemento objeto de búsqueda se encuentra en el conjunto cuando realmente no es así. Esto se debe a que determinados bits han cambiado su valor a nivel alto ('1') durante la inclusión en el conjunto de nuevos elementos. Aunque es una situación a tener en cuenta durante la fase de modelado, la tasa de error típica para este fenómeno suele ser del 1% o, incluso, menor (se incrementa de forma proporcional al número de elementos que integran el conjunto). Por otro lado, se debe indicar que la probabilidad de obtener "falsos negativos" es nula.

La probabilidad de "falsos positivos" viene determinada por la expresión:

$$f = (1 - e^{-\frac{kn}{m}})^k \quad (1)$$

donde $k$ es el número de funciones *hash*, $m$ el número de posiciones del *array* y $n$ el número de *ítems* que integran el conjunto.

Dicho de otra forma, el número $k$ de funciones $hash$ necesarias para obtener una baja tasa de "falsos positivos" dependerá del tamaño $m$ del *array* y de la dimensión $n$ del conjunto:

$$k = \ln(2) \cdot {}^{m}/_{n} \quad (2)$$

Por tanto, tal como se observa en la expresión anterior, la probabilidad de falsos positivos únicamente dependerá de la relación bit por elemento $m/n$.

Un BF en su versión simple permite realizar únicamente dos operaciones: inclusión de elementos en el filtro y consulta sobre si un elemento concreto se encuentra en el

conjunto. De forma inicial, un BF no es más que un vector de $n$ *bits* con todos y cada uno de ellos a nivel bajo ('0'). Cada vez que se realiza una inserción de elementos en el filtro se mapea, mediante funciones *hash* que siguen una distribución aleatoria uniforme, cambiando ciertas posiciones del vector (tantas como funciones *hash* se haya decidido ejecutar) a 1 [72].

Cuando se inserta un elemento en el vector, el *hash* correspondiente cambiará su *bit* de estado a 1. Cada patrón introducido en el conjunto cambiará a 1 tantas posiciones del vector como funciones *hash* hayamos decidido ejecutar.

Si se desea realizar una consulta, habrá que ejecutar de forma paralela las mismas funciones *hash* aplicadas durante el proceso de inserción y comprobar que la operación binaria AND devuelve un 1, lo que supone que todos los bits consultados tendrán también ese valor y que la consulta puede ser satisfactoria o no, debiendo considerar la pequeña probabilidad que existe de que se trate de un falso positivo.

Tal como se ha comentado anteriormente, como los falsos negativos no están permitidos, no hay posibilidad de eliminar elementos del filtro, ya que cambiar a nivel bajo un determinado bit del vector puede significar estar incidiendo sobre más de un elemento, conduciendo a eliminaciones erróneas de elementos. Si se quiere introducir este modo de funcionamiento, se debe implementar la modificación de los *Bloom Filter* con contador (CBF por sus siglas en inglés), que incluye un vector de contadores con cada una de sus posiciones asociadas a una de las del BF original; de esta forma, los contadores se incrementan o decrementan en función de si se añaden o se eliminan, respectivamente, elementos en el BF que se mapeen en esas posiciones en concreto (Figura 26).

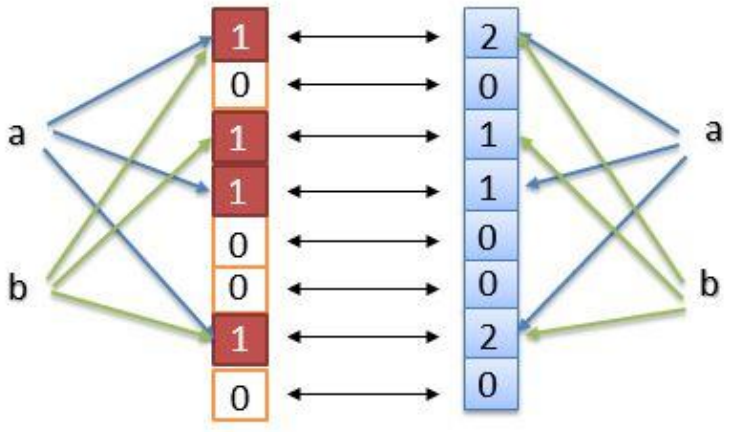



**Figura 26. Ejemplo de CBF con dos elementos.**

Esta modificación de los BF es la más empleada en la actualidad para monitorizar y direccionar en redes de comunicaciones debido a sus buenas prestaciones [73], por lo que será la que implementaremos en los filtros de análisis de los paquetes de red.

#### *3.2.1.2. Función hash H3*

Por otra parte, es necesario explicar cómo actúa la función *hash* implementada para interactuar con los CBF: la función *hash* universal H3. Esta función es idónea para una implementación *hardware* como la que se ha realizado en este trabajo, al estar basada únicamente en operaciones AND bit a bit (y su acumulación iterativa mediante OR exclusiva o XOR) entre aquel elemento objeto de la operación, ya sea una dirección IPv6 o un puerto UDP, y una matriz previamente definida con la misma dimensión que el elemento a gestionar [74]. A partir de los tests realizados, se ha podido concluir que se trata de una opción bastante eficiente en cuanto a tiempo de ejecución y consumo de recursos se refiere.

$$h(x) = x_1 \cdot q_1 \oplus x_2 \cdot q_2 \oplus \ldots \oplus x_i \cdot q_i \tag{3}$$

A continuación, se muestra el código realizado para su modelado, en concreto para el caso de una dirección IPv6 (para los puertos UDP presenta la misma estructura, variando únicamente la longitud de los parámetros de la función y, por tanto, el número de iteraciones a realizar en el bucle).

```
void hash_func_ip(ap_uint<8> ip_to_hash[IPv6_ADDR_SEGMENT],
            ap_uint<8> matrix_ip[MATRIX_SIZE_IP]) {

      hash_ip = 0;
      for (i = 0; i < IPv6_ADDR_SEGMENT; i++) {
            hash_ip ^= (ip_to_hash[i] & matrix_ip[i]);
      }

#if DEBUG
      printf("The hash_ip value is: %u \n", hash_ip.VAL);
#endif
}
```

### 3.2.2. Creación de filtros individuales para niveles de red y transporte

En este apartado se procederá a describir el proceso de diseño de los bloques IP encargados de filtrar, en función de una *blacklist* previamente definida que recoja dispositivos que sean considerados como una posible amenaza para el sistema en el caso de

direcciones IPv6 de origen que intenten establecer conexiones con el *Fog Node*, o en función de una *whitelist* en la que se definan los puertos del nodo a través de los que se admitan conexiones entrantes para el caso de UDP, desestimando de esta forma intentos de accesos a otros puertos que puedan comprometer la integridad del *Fog Node* (Figura 27). De esta forma, se consigue optimizar el cómputo de nuestro sistema, así como el número de accesos a memoria realizados (principal *bottleneck* en la mayoría de los sistemas electrónicos empotrados).

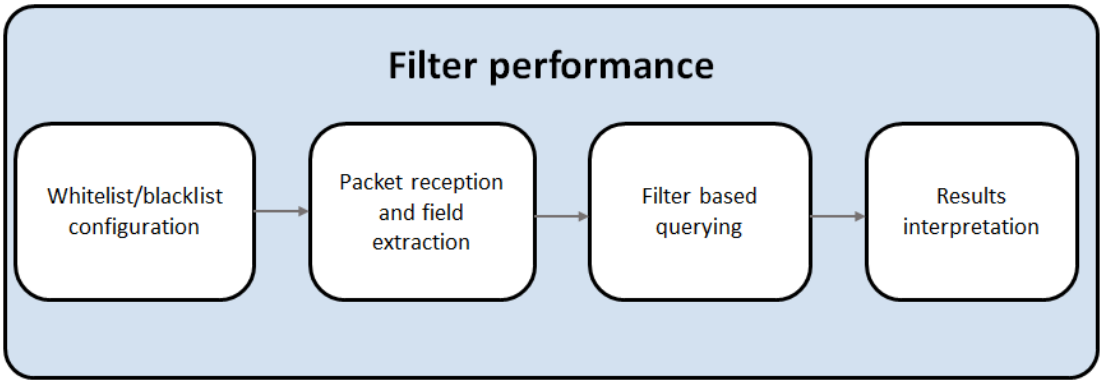



**Figura 27. Flujo de trabajo de los filtros para IoT.**

Un detalle a señalar de forma previa a definir el proceso de diseño es que, a pesar de que las direcciones IPv6 están compuestas por 128 bits, debido a los problemas que presenta Vivado HLS para gestionar tipos de datos mayores de 64 bits (máximo tamaño estandarizado, a pesar de que admiten precisiones de hasta 1024 bits para sus tipos de datos propios `ap_(u)int<precisión>`) se ha optado, finalmente, por gestionar las direcciones origen como dos bloques bien diferenciados de 64 bits. Esto es posible debido al formato que presentan las direcciones en IPv6, estando compuestas por un bloque inicial de 64 bits que identifica la red y que es empleado para tareas de encaminamiento, y un segundo fragmento de la misma dimensión que permite identificar a un determinado nodo dentro de esa red.

Como se ha indicado anteriormente, estos filtros están basados en una estructura CBF modelada en C/C++ [68].

### *3.2.2.1. Modelado de la funcionalidad*

Más allá de la etapa de inicialización de los distintos parámetros de que consta el BF (capacidad, tasa de falsos positivos, dimensión del *array*, número de elementos actuales,

etc.) se puede considerar, de forma resumida, que la funcionalidad de los filtros presenta dos partes bien diferenciadas: por un lado, aquellas tareas vinculadas a la actividad del BF (inserción y eliminación de direcciones IPv6 o puertos UDP para gestionar las listas de acceso al sistema, y la operación de consulta) y, por otro, las asociadas a la gestión de los contadores, existiendo una función asociada a cada una de las tareas del BF (incremento de contadores a inserción de elementos, decremento de contadores a eliminación de elementos y comprobación de contadores a consulta sobre una dirección IPv6 o un puerto UDP concreto).

Los métodos de inclusión y eliminación de elementos presentan una estructura bastante similar. Cambia únicamente la operación asociada que realizan sobre los contadores (incremento o decremento, respectivamente). De forma inicial, se les pasa como parámetro de entrada el fragmento de 64 bits de la dirección a gestionar en el caso de IPv6 o el puerto en cuestión sobre el que se va a realizar la operación en el caso de UDP.

Una vez que se cuenta con estos parámetros, se genera en primer lugar un *offset* que regirá la posición de los índices en los que se mapeará un elemento en concreto. Esta operación se realiza aplicando operaciones OR bit a bit entre una variable previamente definida de la misma dimensión que los elementos tratados y su posterior acumulación mediante XOR. Finalmente, para evitar que se genere un *offset* de gran tamaño (especialmente al gestionar los fragmentos de 64 bits de las direcciones IPv6), el valor final será la mitad del obtenido mediante la aplicación de las operaciones binarias mencionadas:

```
void seed_generation_ip(ap_uint<8> ip_address_field[IPv6_ADDR_SEGMENT]) {

	offset = 0;
	for (i = 0; i < IPv6_ADDR_SEGMENT; i++) {

		offset ^= seed_ip[i] | ip_address_field[i];
		printf("seed: %u , ip_address_field: %u \r\n", (short) seed_ip[i],
				(short) ip_address_field[i]);
	}
	offset = offset / 2;

	printf("The generated offset for ip field is: %u \n", offset);
}
```

Posteriormente, es el turno de aplicar la función *hash* H3 al elemento a gestionar, proporcionándole a dicha función, además del valor de este elemento, el de la matriz que se empleará para realizar las operaciones binarias comentadas. Una vez que se dispone del

valor tanto del *hash* como del *offset*, se generan tantos índices como funciones *hash* sea necesario aplicar para asegurar la tasa de falsos positivos definida (en este caso, 4 índices para asegurar una tasa de falsos positivos del 0,1%). Tras obtener los índices, se incrementan o decrementan, en función de la operación aplicada, los contadores asociados a las posiciones cuyo valor es la mitad de los índices generados (cálculo realizado para que el funcionamiento del BF no sea tan evidente). Además, en caso de que el índice generado sea mayor al doble del tamaño del *array* definido para el BF, se define su valor como el máximo índice posible (al haberse definido el *array* del BF con 250 posiciones, el índice máximo posible será 500).

A continuación, se incluye la implementación del método de inserción para el caso de IPv6, junto con la consiguiente operación asociada de incremento de los contadores para los índices generados:

```
ap_uint<1> ip_field_add(ap_uint<8> ip_add[IPv6_ADDR_SEGMENT]) {

	cbf_index = 0;

	seed_generation_ip(ip_add);
	hash_func_ip(ip_add, matrix_ip);

	for (i = 0; i < ip_filter.nfuncs; i++) {
		cbf_index = hash_ip + offset * i;
		if (cbf_index > MAX_INDEX) {
			cbf_index = MAX_INDEX;
			bitvector_increment(cbf_index);
		} else {
			bitvector_increment(cbf_index);
		}
#if DEBUG
		printf("The generated index is: %u \n", cbf_index.VAL);
#endif
	}

	return 0;
}
```

```
ap_int<1> bitvector_increment(ap_uint<10> index) {

	position = 0;
	if (index % 2 != 0) {
		position = (index / 2) + 1;
	} else {
		position = index / 2;
	}
	ap_uint<4> n = ip_filter.array[position];
	if (n == 0x0f) {
```

```
            printf("Error, 4 bit counter overflow\n");
            return -1;
      }

      n = n + 1;

      ip_filter.array[position] = n;
      return 0;
}
```

Por otro lado, tenemos el método de consulta, que difiere básicamente de los anteriores, además de en la operación de contadores que tiene asociada, en que el elemento sobre el que se actúa ha sido extraído directamente de un paquete de red concreto en los campos de la cabecera oportunos en cada caso y no proporcionado por el diseñador para configurar las políticas de privacidad del nodo. El método de extracción tanto de las direcciones IPv6 como de los puertos UDP será comentado en el siguiente apartado. A continuación, se muestra la tarea de consulta, en concreto la incluida en el filtro UDP:

```
ap_uint<1> udp_port_query(ap_uint<8> extracted_port[UDP_ADDR_SIZE]) {

      cbf_index = 0;
      status = 0;

      seed_generation_udp(extracted_port);
      hash_func_udp(extracted_port, matrix_udp);

      for (i = 0; i < udp_filter.nfuncs; i++) {
            cbf_index = hash_udp + offset * i;
#if DEBUG
            printf("The generated index is: %u \n", cbf_index.VAL);
#endif
            if (!(bitvector_check(cbf_index))) {
                  status = 0;
            } else {
                  status = 1;
                  break;
            }

      }
      return status;
}
```

Además, en el filtro para UDP se ha incluido una función, denominada *transport_protocol*, que se encarga de extraer el campo *Next Header* de la cabecera IPv6 y que permite identificar qué protocolo de la capa de transporte ha sido empleado en la comunicación establecida. De esta forma, en caso de que el protocolo definido no sea UDP (valor 0x11 en hexadecimal, 17 en decimal), automáticamente se descarta el filtrado a este

nivel del modelo OSI, ahorrando de esta forma ciclos de ejecución del sistema y, por tanto, reduciendo la latencia del mismo.

```
ap_uint<2> transport_protocol(ap_uint<32> data_in) {

	static ap_uint<2> result = 2;
	ap_uint<8> next_header = 0;

	next_header = data_in.range(31, 24);

	if (next_header == 0x11) {
		result = UDP_DETECTION;
	} else {
		result = NOT_UDP;
	}

	return result;
}
```

### *3.2.2.2. Definición de la función top y de las interfaces de comunicación*

Tras haber modelado en alto nivel la funcionalidad que deben incluir ambos filtros para cumplir con los objetivos propuestos, se hace necesario posteriormente definir una función *top* para cada uno de ellos, que controle la operación a realizar en cada momento y sobre la que se definan las interfaces de comunicación que deben incluir dichos filtros.

Las operaciones generales a realizar, tal como se ha indicado anteriormente son, además de la inicialización del BF, la inserción y eliminación de elementos, así como la búsqueda de campos específicos dentro de una cabecera Ethernet (identificadores de red y de nodo origen para IPv6, puertos UDP de destino a nivel de la capa de transporte).

La extracción de los campos oportunos de la cabecera se realiza conociendo las palabras de 32 bits (ya que ha sido el tamaño de palabra definido para transmitir las distintas tramas de que consta un paquete de red completo) en las que dicha información está ubicada para, posteriormente, obtener el rango de bits concreto que se precisa para conocer esta información. Para ello, no se ha considerado registrar la cabecera (parte de interés del paquete Ethernet durante la etapa de filtrado) y se iterará directamente sobre la interfaz AXI4-Stream de entrada, evitando de esta forma consumir ciclos de reloj adicionales.

Por otro lado, para que los bloques de filtrado ejecuten correctamente su tarea, en principio, no es necesario gestionar la señal TLAST de la interfaz AXI4-Stream, ya que este indica el final de un paquete y, por lo general, en Ethernet el *payload* puede estar vacío,

pero siempre se debe contar con la cabecera. De lo contrario, el paquete nunca será direccionado hacia su destino. La iteración sobre el paquete finalizará cuando se haya alcanzado la última palabra de 32 bits donde se encuentra información útil correspondiente al campo analizado, con el fin de no incidir en aumentos innecesarios de la latencia del bloque.

A continuación, se muestra la estructura de la función *top* para el filtro de UDP, siendo equivalente para el filtro IPv6, diferenciándose únicamente en las interfaces de comunicación empleadas (que serán comentadas más adelante) y en que en la función de búsqueda es necesario extraer dos campos distintos: el del identificador de red y, en caso de obtener un positivo en dicha operación, el del identificador de nodo.

```
ap_uint<1> udp_cbf(ap_uint<3> mode, ap_uint<16> udp_port,
            ap_axis<32, 1, 1, 1> * packet) {

#pragma HLS INTERFACE s_axilite port=udp_port bundle=axi_lite
#pragma HLS INTERFACE s_axilite port=mode bundle=axi_lite
#pragma HLS INTERFACE s_axilite port=return bundle=axi_lite name=status
#pragma HLS INTERFACE axis register both depth=400 port=packet

      ap_uint<6> counter;
      static ap_uint<2> detection;
      ap_uint<1> in_last;
      ap_uint<32> in_data;

      ap_uint<8> udp_tmp[UDP_ADDR_SIZE];
      ap_uint<8> config_udp[UDP_ADDR_SIZE];

      switch (mode) {

      case INIT:
            udp_filter_init(CAPACITY, ERROR_RATE);
            status = 0;
            break;

      case ADD:

            config_udp[0] = 0;
            config_udp[1] = 0;

            config_udp[0] = udp_port.range(15, 8);
            config_udp[1] = udp_port.range(7, 0);

            status = udp_port_add(config_udp);
            break;

      case REMOVE:

            config_udp[0] = 0;
            config_udp[1] = 0;
```

```
		config_udp[0] = udp_port.range(15, 8);
		config_udp[1] = udp_port.range(7, 0);

		status = udp_port_remove(config_udp);
		break;

	case SEARCH:

		in_last = 0;
		in_data = 0;
		detection = 2;

		udp_tmp[0] = 0;
		udp_tmp[1] = 0;

		counter = 0;

		do {
			in_data = packet[counter].data;

			if (counter == 5) {
				in_data = packet[counter].data.range(31, 0);
				detection = transport_protocol(in_data);
				printf("Transport protocol status: %u \r\n", (short)
detection);

				if (detection != 0) {
					status = 1;
					break;
				}
			}

			if ((counter == 14) && (detection == UDP_DETECTION)) {
				udp_tmp[0] = in_data.range(31, 24);
				udp_tmp[1] = in_data.range(23, 16);

				status = udp_port_query(udp_tmp);
				printf("UDP port search status: %u \r\n", (short)
status);
				break;
			}
			counter++;
		} while (counter != HEADER_SIZE);

		break;

	default:
		break;
	}
	return status;
}
```

En cuanto a las interfaces de comunicación, tres de ellas son compartidas por los dos bloques IP diseñados:

a. *Mode.* Su valor determina qué operación de las incluidas en el filtro (inicialización, inserción, eliminación o consulta de elementos) se debe realizar en cada momento.
b. *Packet.* Paquete de datos a analizar por el filtro y del cual se deben extraer los campos oportunos de la cabecera para su análisis en función de la misión de cada filtro.
c. *Status.* Devuelve una respuesta sobre si la operación a realizar y especificada por la interfaz de entrada *mode* se ha llevado a cabo de forma satisfactoria o no.

Además de estas dos interfaces comunes, el filtro IPv6 incluye tres interfaces de entrada adicionales:

a. *Ip_segment_1* e *ip_segment_2.* Segmentos de 32 bits que conforman el fragmento final de 64 bits (identificador de red o de nodo) de la dirección IPv6 origen a incluir o eliminar de la *blacklist* que gestiona el filtro.
b. *Field_type.* Permite identificar si el segmento de 64 bits pasado a través de las interfaces de entrada *ip_segment_1* e *ip_segment_2* se corresponde con un identificador de red o de nodo.

Por su parte, el filtro de UDP sólo incluye una interfaz de entrada adicional:

a. *UDP_port.* Identifica el puerto UDP a incluir o quitar de la *whitelist* de conexiones permitidas del *Fog Node*.

En referencia a su definición, se ha optado por definir tanto *mode* como las interfaces de entrada propias de cada uno de los dos filtros como interfaces AMBA AXI4-Lite esclavas, permitiendo de esta forma que funcione como un puerto de configuración (sólo permite una transferencia por transacción) que interactúa con el diseñador para transmitir los valores oportunos para el correcto funcionamiento del sistema, tanto para la gestión de las listas de acceso como para definir la operación a realizar. En este sentido, debido a que el ancho máximo del campo de datos de AXI4-Lite para los dispositivos Zynq actualmente se mantiene como 32 bits (para otros dispositivos como UltraScale+ o Virtex-7 se puede definir este ancho como 32 o 64 bits haciendo uso en Vivado del bloque AXI SmartConnect), es necesario pasar los fragmentos IPv6 de 64 bits en dos tramas de 32 bits y concatenarlos posteriormente dentro de la llamada a cada uno de los modos de la función *top*.

La interfaz de salida `status` también se define con el mismo tipo de interfaz AMBA AXI4-Lite, con el objetivo de transmitir al PS el valor del resultado de la operación realizada por el filtro. Todas estas interfaces AXI4-Lite se incluyen en un mismo bus, ya que presenta señales independientes para los procesos de lectura y escritura, así como las señales TVALID y TREADY para informar del estado de los extremos que están inmersos en el proceso de comunicación.

Por su parte, la interfaz de entrada `packet` se ha definido como una interfaz AMBA AXI4-Stream, que permite un flujo de información entrante a través de la red a la que se encuentra conectado el dispositivo, permitiendo ráfagas de datos sin limitaciones por cada transacción realizada a altas tasas de transferencia. Es necesario indicar en este punto la profundidad de dicha interfaz, la cual se ha definido como 400 al ajustarse a la cantidad de palabras de 32 bits en las que se divide un paquete Ethernet de tamaño máximo (1500 *bytes*), a pesar de que la parte de interés para este trabajo es la cabecera del paquete y el *payload* no es analizado.

En la Figura 28 se muestra el diagrama de entrada/salida para los filtros modelados:

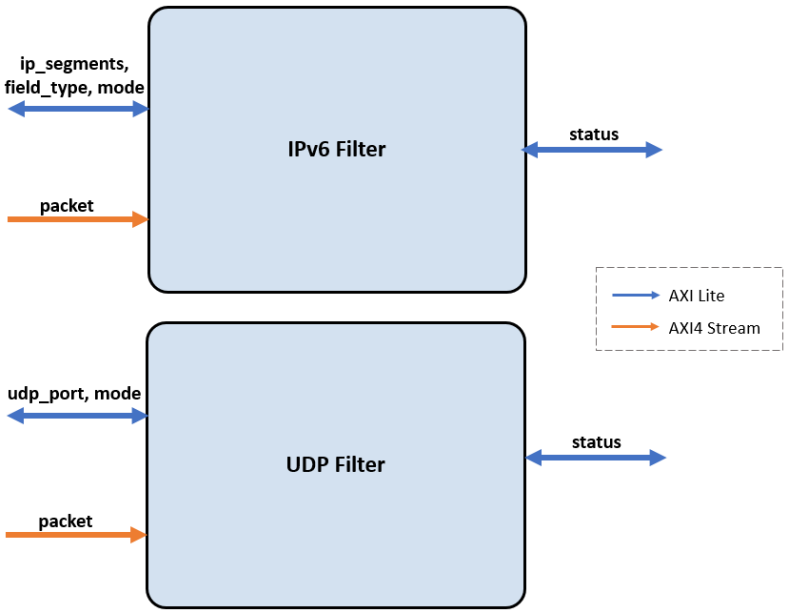



**Figura 28. Diagramas de E/S de los filtros individuales.**

### *3.2.2.3. Verificación*

Para comprobar la funcionalidad de los filtros, se hace necesario el diseño de un banco de pruebas o *testbench* que permita cubrir los distintos casos, especialmente aquellos más críticos, que podrían darse durante la ejecución del sistema, con el fin de evitar la

propagación de errores hacia etapas posteriores del flujo de diseño, lo que conlleva una mayor dificultad para su detección y eliminación.

Centrándonos en este caso particular, el *testbench* está compuesto por:

- Definición de los elementos a incluir o eliminar de los BF.
- Operaciones propias de los filtros como son la inclusión y la eliminación de elementos.
- Tareas de consultas sobre paquetes previamente definidos y de los que se conoce el valor de la dirección IPv6 origen y del puerto UDP destino para poder contrastar los resultados obtenidos.

A continuación, se muestra un extracto del *testbench* empleado para verificar el correcto funcionamiento del filtro para puertos UDP:

```
int test_cbf() {

	int i = 0;
	ap_uint<16> port_1 = 33435;
	ap_uint<16> port_2 = 22;
	ap_uint<16> port_3 = 72508;
	ap_uint<16> port_4 = 5144;

	for (i = 0; i < HEX_WORDS_LIMIT; i++)
		packet[i].data = 0;

	ap_uint<1> status_t = 2;
	short elements_added = 0, elements_removed = 0, elements_included = 0;

	printf("\n Test running\n");
	//Filter initiation with specific parameters
	status_t = udp_cbf(INIT, port_4, packet);

	if (status_t) {
		printf("ERROR: The filter couldn't be created.\n");
		return 1;
	}
	printf("\n UDP filter was created.\n");
	//Ports insertion
	printf("UDP ports addition to the filter.\n");
	status_t = udp_cbf(ADD, port_1, packet);
	printf("status: %u\r\n", (int) status_t);
	elements_added++;
	status_t = udp_cbf(ADD, port_2, packet);
	printf("status: %u\r\n", (int) status_t);
	elements_added++;

	//Delete specific ports from the filter
	printf("UDP ports elimination from the filter.\n");
	status_t = udp_cbf(REMOVE, port_2, packet);
	printf("status: %u\r\n", (int) status_t);
```

```
	elements_removed++;
	status_t = udp_cbf(REMOVE, port_4, packet);
	printf("status: %u\r\n", (int) status_t);
	elements_removed++;

	//Request about if an udp_port are in a specific payload or not (OK SEARCH)
	printf("Search patterns in the payload. \n");
	configPayload(packet);
	status_t = udp_cbf(SEARCH, port_4, packet);
	printf("status: %u\r\n", (int) status_t);
	if (status_t == 0) {
		printf("The UDP port was found\r\n");
		elements_included++;
		printf("Elements found : %u \r\n", elements_included);
		printf("status: %u\r\n", (int) status_t);
	} else {
		printf("There aren't coincidences for any parameter\r\n");
	}

	//Delete the correct port from the filter
	printf("UDP port elimination from the filter.\n");
	status_t = udp_cbf(REMOVE, port_1, packet);
	printf("status: %u\r\n", (int) status_t);
	elements_removed++;
	status_t = 2;

	 printf("status: %u\r\n", (int) status_t);

	//Request about if an udp_port are in a specific payload or not (NEGATIVE
SEARCH)
	printf("Search patterns in the payload. \n");
	configPayload(packet);
	status_t = udp_cbf(SEARCH, port_4, packet);
	printf("status: %u\r\n", (int) status_t);
	if (status_t == 0) {
		printf("The UDP port was found\r\n");
		elements_included++;
		printf("Elements found : %u \r\n", elements_included);
		printf("status: %u\r\n", (int) status_t);
	} else {
		printf("There aren't coincidences for any parameter\r\n");
	}

	printf("Elements found : %u \r\n", elements_included);

	//Results
	printf("Final statistics: \n"
			"Elements added:   %d" "\n"
			"Elements deleted: %d" "\n"
			"Elements found: %d" "\n", elements_added, elements_removed,
			elements_included);

	return 0;
}
```

### *3.2.2.4. Síntesis de alto nivel*

A continuación, es turno de llevar a cabo el proceso de síntesis de alto nivel para obtener la descripción RTL equivalente al código modelado en C/C++ y verificar que se cumplen las restricciones temporales especificadas de forma inicial, amén a un consumo razonable de recursos lógicos en la FPGA, ya que el sistema completo estará compuesto por más bloques IP.

De forma previa a la realización de esta etapa del flujo de diseño, se hace necesario un análisis sobre qué directivas de optimización se pueden aplicar para mejorar las prestaciones del bloque IP. En este caso, sólo se ha optado por aplicar la directiva UNROLL para realizar el desenrrollado de ciertos bucles de *reset* (asignación a vectores del valor 0 en todas sus posiciones). Se ha estudiado la posibilidad de aplicar *pipelining* en algunos de los bucles existentes en la descripción; sin embargo, las dependencias de datos y las operaciones de lectura y escritura sobre una misma variable imposibilitan la inclusión de dicha directiva.

Aplicando como restricción temporal la frecuencia de 200 MHz (5 ns de periodo de reloj), se obtienen los resultados reflejados en la Tabla 2 y en la Tabla 3 tanto para *timing* como para consumo de recursos lógicos, respectivamente. Indicar que Vivado HLS introduce el concepto de incertidumbre para tener un margen controlado para el retardo introducido por las interconexiones en el dispositivo. Si no se le asigna un valor, asume el 12.5% del periodo de reloj.

**Tabla 2. Resultados temporales en síntesis de alto nivel de los filtros individuales.**

| | IPv6 | UDP |
|---|---|---|
| **Periodo de reloj objetivo** | 5,00 ns | 5,00 ns |
| **Periodo de reloj estimado** | 3,89 ns | 3,31 ns |
| **Incertidumbre** | 0,56 ns | 0,46 ns |

**Tabla 3. Consumo de recursos en síntesis de los filtros individuales.**

| | IPv6 | UDP |
|---|---|---|
| **BRAM_18K** | 2/280 (<1%) | 1/280 (<1%) |
| **DSP48E** | 0/220 (0%) | 0/220 (0%) |
| **FF** | 2.475/106.400 (2%) | 854/106.400 (<1%) |
| **LUT** | 3.771/53.200 (7%) | 1.814/53.200 (3%) |

Como se puede observar, se obtiene para ambos casos periodos de reloj estimados por debajo de los 5 ns predefinidos y, por tanto, obteniendo frecuencias de trabajo por encima de los 200 MHz establecidos como requisito de operación (257 MHz para IPv6 y 302 MHz para UDP).

Por otra parte, los consumos de recursos lógicos son lo suficientemente reducidos en comparación con el total, por lo que se dan estos valores como aceptables para continuar con la etapa de cosimulación.

### *3.2.2.5. Cosimulación y creación del bloque IP*

Generada la descripción RTL, es preciso verificar que el código HDL equivalente obtenido se comporta de la misma forma que su homólogo en alto nivel. Para ello, se cuenta en la herramienta de síntesis de alto nivel con la etapa de cosimulación en la cual, a partir del *testbench* original diseñado en C/C++, se genera un test equivalente a nivel RTL que permite verificar el correcto comportamiento de la descripción *hardware*, ejecutando ambas pruebas de forma concurrente con el fin de comparar las respuestas generadas.

Hay que señalar que la descripción RTL se puede generar en el lenguaje HDL que se considere oportuno por parte del diseñador. En este caso, se ha elegido Verilog debido al buen rendimiento que han tenido las implementaciones en este lenguaje en proyectos anteriores realizados en la división. Finalmente, tal como se observa en la Tabla 4, las descripciones RTL generadas para ambos filtros superan con éxito los *testbenches* equivalentes generados. Es preciso indicar que el valor medio incluido no se trata de la media aritmética de los valores mínimos y máximos obtenidos, sino que es el número de ciclos de reloj que el bloque IP toma de media para completar la ejecución de las operaciones incluidas en el *testbench*.

**Tabla 4. Resultados de cosimulación para los filtros individuales.**

| | | | Latencia | | | Intervalo | | |
|---|---|---|---|---|---|---|---|---|
| **Protocolo** | RTL | Status | Mínimo | Media | Máximo | Mínimo | Media | Máximo |
| **IPv6** | Verilog | Pass | 62 | 195 | 205 | 52 | 185 | 195 |
| **UDP** | Verilog | Pass | 46 | 113 | 181 | 38 | 99 | 173 |

En lo que respecta a la latencia se puede observar que, para el caso del filtro de IPv6, se obtiene un número medio de ciclos de reloj de 195 para completar las funciones especificadas en su respectivo *testbench*, lo que da como resultado, teniendo en cuenta el tiempo de ciclo aproximado de 3,89 ns obtenido en la etapa de síntesis de alto nivel, una latencia total de 0,7585 μs. Por su parte, el filtro UDP, que realiza menos operaciones en su *testbench* que su homólogo para IPv6 (tiene menos casos que cubrir), obtiene un número medio de ciclos de reloj de 113 que, teniendo en cuenta el tiempo de ciclo de 3,31 ns, da como resultado una latencia total de 0,3740 μs.

En cuanto al intervalo, parámetro que refleja el número de ciclos de reloj en los que el bloque IP puede recibir nuevos datos antes de terminar con la operación inmediatamente anterior, vemos que se observa una diferencia de 10 ciclos respecto a los valores de latencia para IPv6 y de entre 8 y 14 en el caso de UDP, lo que refleja que, para ambos filtros, hay ciertas etapas del código que se están ejecutando concurrentemente, dando lugar al paralelismo buscado a la hora de implementar funcionalidad concreta en *hardware* y, más concretamente, en una FPGA.

En la Figura 29 se puede observar el diagrama de formas de onda para el filtro UDP en el que, tras configurar el puerto de destino que se conoce previamente que incluye el paquete de prueba, se detecta que el protocolo de transporte es UDP y se pasa a realizar la extracción del campo en la palabra de 32 bits correspondiente, obteniendo una búsqueda positiva y retornando la variable `status`, por tanto, el valor 0. Durante el transcurso del *testbench*, se observa un caso de búsqueda incorrecta (`status` con valor 1), el cual ha sido forzado eliminando del CBF el puerto coincidente de forma previa a la realización de esta operación.

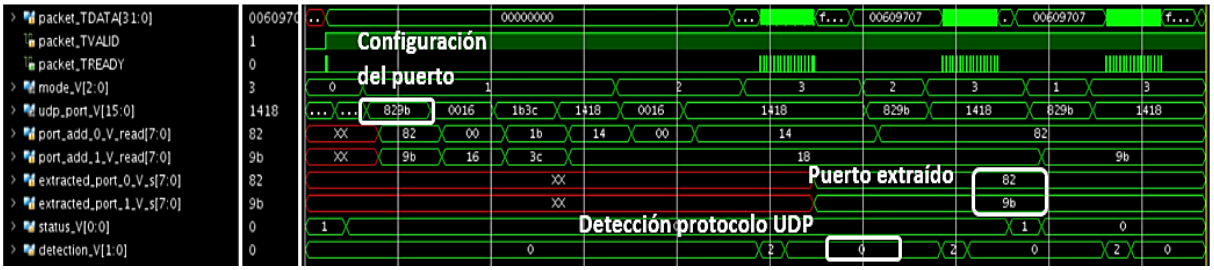



**Figura 29. Resultados de cosimulación para el filtro UDP.**

Del mismo modo, en la Figura 30 y la Figura 31 se muestra la ejecución de una búsqueda completa (tanto red como nodo) por parte del filtro para IPv6, resaltando que, en ambos casos, la consulta coincide con una dirección previamente configurada en la *blacklist* del propio filtro y devolviendo la variable `status`, por tanto, el valor 0.

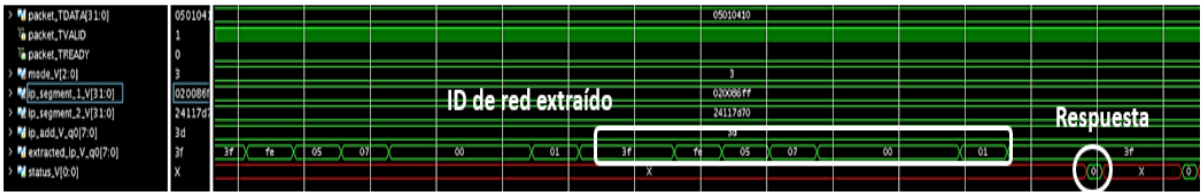



**Figura 30. Resultados de cosimulación para búsqueda de red en IPv6.**

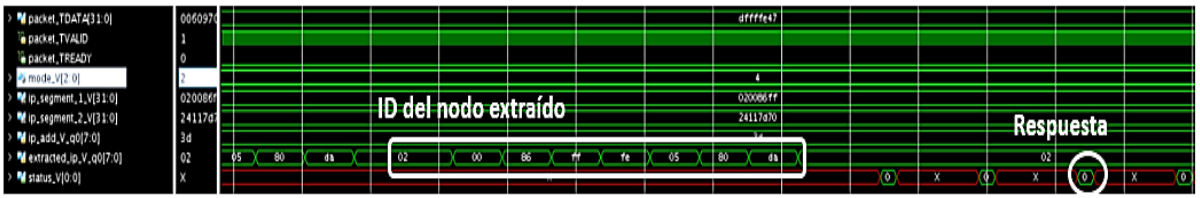



**Figura 31. Resultados de cosimulación para búsqueda de nodo en IPv6.**

Finalmente, como paso final de la etapa de desarrollo en alto nivel, es necesario exportar cada uno de los filtros como un bloque IP, de forma que pueda ser integrado en una plataforma definida mediante un diagrama de bloques en el Vivado IDE, conformando de este modo el sistema final. Para ello, una vez que se hace uso de la herramienta de exportación que incluye Vivado HLS, es recomendable evaluar nuevamente la descripción RTL generada para obtener resultados finales tanto de latencia como de consumo de recursos lógicos más reales ya que, durante la etapa de síntesis de alto nivel, no se tienen en cuenta las restricciones impuestas por las interfaces de comunicación definidas, especialmente por el puerto de configuración AXI4-Lite, que trabaja a un *throughput* comprendido entre 150 y 200 MHz.

A partir de los resultados reflejados en la Tabla 5 y en la Tabla 6 vemos como, para ambos filtros, los resultados de *timing* empeoran sustancialmente, pero aun así se mantienen por debajo de los 5 ns definidos como requisito de funcionamiento de los bloques IP. Por otro lado, se observa, también para ambos filtros, un descenso considerable del número de recursos lógicos consumidos para su implementación, especialmente en el caso de los *flip-flops* y las LUTs.

**Tabla 5. Resultados temporales finales obtenidos para los bloques los filtros individuales.**

| | IPv6 | UDP |
|---|---|---|
| **Periodo de reloj objetivo** | 5,000 ns | 5,000 ns |
| **Periodo de reloj conseguido** | 4,364 ns | 3,606 ns |

**Tabla 6.Consumo de recursos obtenidos para los bloques de los filtros individuales.**

| Recursos | IPv6 | UDP |
|---|---|---|
| **BRAM** | 2 | 1 |
| **DSP** | 0 | 0 |
| **FF** | 1.251 | 601 |
| **LUT** | 626 | 539 |

## 3.3. Conclusión

En este capítulo se ha descrito el proceso de diseño del subsistema de filtrado del sistema de seguridad completo a implementar para un nodo *Fog*, haciendo uso de la herramienta de síntesis de alto nivel Vivado HLS. Después de haber introducido los conceptos preliminares necesarios para comprender la estructura y el modo de operar de los filtros, se ha pasado a describir con detalle el modelado de su funcionalidad, incluyendo el código de aquellas funciones más relevantes, el modo de verificación y las interfaces de comunicación definidas.

Finalmente, se aportan los valores obtenidos en términos de latencia y de consumo de recursos. Asimismo, es necesario resaltar que la inclusión de las interfaces de comunicación en el diseño se produce durante la fase de exportación del IP al flujo de diseño de síntesis lógica, motivo por el cual los resultados temporales obtenidos durante la etapa

de síntesis de alto nivel son más optimistas, al no tener en cuenta las limitaciones impuestas por las interfaces, las cuales suelen ser parte de las rutas críticas del bloque IP.

# Capítulo 4. Diseño de la etapa de cifrado

## 4.1. Introducción

Tras hacer frente en el capítulo anterior al diseño completo y optimización del bloque IP con la funcionalidad necesaria para realizar un proceso de autenticación de las conexiones entrantes al nodo *Fog*, a continuación se describen las alternativas analizadas para implementar las condiciones necesarias de cifrado de la información para asegurar las comunicaciones entre el sistema y los equipos externos que intentan acceder a los dispositivos IoT bajo su rango de actuación.

En este caso, se ha planteado estudiar tanto la librería TweetNaCl como el bloque de cifrado Simon por su relación ocupación de memoria/seguridad, exponiendo en los siguientes apartados sus ventajas y desventajas, así como la solución que finalmente se ha adoptado.

## 4.2. Análisis de la librería criptográfica TweetNaCl

Tal como se comentó en el apartado de Librerías criptográficas, TweetNaCl se plantea como una opción idónea para las operaciones de cifrado a realizar en una implementación *hardware* como la que nos incumbe en este trabajo, debido a su reducido número de líneas de código (que derivan en un uso reducido de memoria) y a sus altas prestaciones en cuanto a velocidad y seguridad se refiere.

### 4.2.1. Perfilado de la librería empleando Valgrind

En primer lugar, se hace necesario conocer qué operaciones de las incluidas en la librería precisan de una mayor carga computacional para su implementación y, por tanto,

son candidatas a ser aceleradas en *hardware*, ya que trasladar toda la funcionalidad no es una opción viable en términos de latencia y consumo de recursos (la demanda de DSPs que requiere una librería criptográfica como esta es mayor que la cantidad disponible en una FPGA como la que integra el dispositivo Zynq).

Para ello, se ha realizado un perfilado *software* de una aplicación que incluya las operaciones básicas para cumplir con la funcionalidad de cifrado/descifrado deseada. La aplicación en cuestión incluye la generación de claves públicas a partir de las privadas de cada extremo de la comunicación, así como las funciones específicas de cifrado y descifrado sobre un mismo mensaje. Adicionalmente, se incluyen una serie de etapas de control de errores para comprobar que la aplicación se está llevando a cabo de forma satisfactoria.

```
int main() {

	int i, j = 0;
	short status = 5;

	/**Randomize once. KEYS: 32 bytes, NONCE: 24 bytes**/
	unsigned char ask[crypto_box_SECRETKEYBYTES] = { 0x67, 0xc6, 0x69,
			0x51, 0xff, 0x4a, 0xec, 0x29, 0xcd, 0xba, 0xab, 0xf2,… ,
		       0x9f, 0xc9, 0x9a};

	unsigned char bsk[crypto_box_SECRETKEYBYTES] = { 0x66, 0x32, 0x0d,
			0x31, 0x58, 0xa3, 0x5a, 0x25, 0x5d, 0x05, 0x17, 0x58,…,
		       0x3d, 0xdc, 0x87};

	 unsigned char nonce[crypto_box_NONCEBYTES] = { 0x69, 0x69, 0x6e,
			0xb6, 0x2b, 0x73, 0xcd, 0x62, 0xbd,…,0x7a, 0x0b, 0x37};


// Data packet definition to make tests (C API requires first 32 bytes to be 0)
	unsigned char message[FRAME_SIZE] = { 0, 0,…, 0, 0, 0x20, 0x60,
			0x97, 0x07, 0x69, 0xea, 0x01,…,0x14, 0x82, 0x8b };

	unsigned char apk[crypto_box_PUBLICKEYBYTES];
	unsigned char bpk[crypto_box_PUBLICKEYBYTES];
	unsigned char ciphertext[FRAME_SIZE];
	unsigned char decrypted[FRAME_SIZE];

	xil_printf("\n Test running...\n");

	/*Public keys generation and exchange*/
	status = crypto_box_keypair(apk, ask);

	if (status != 0) {
	xil_printf("ERROR: The sender public key couldn't be generated!! \r\n");
	}

	xil_printf("The sender public key was generated \r\n");

	status = crypto_box_keypair(bpk, bsk);
```

```
	if (status != 0) {
	xil_printf("ERROR: The receiver public key couldn't be generated!! \r\n");
	}

	xil_printf("The receiver public key was generated \r\n");

	for (i = 0; i < FRAME_SIZE; i++)
		ciphertext[i] = 0;

	/*Encryption process*/
	status = crypto_box(ciphertext, message, (unsigned long long)
FRAME_SIZE,nonce, bpk, ask);

	if (status != 0) {
	xil_printf("ERROR: The frame has not been correctly encrypted!!\n");
		return 1;
	}

	for (i = 0; j < crypto_box_BOXZEROBYTES; j++) {
		xil_printf("Ciphertext value: %x\r\n", ciphertext[i]);
		if (ciphertext[j] != 0) {
		xil_printf("ERROR: the first 16 bytes of the ciphertext are not 0
(incorrect padding)!!\r\n");
	return 1;
		}
	}
	for (i = 16; i < FRAME_SIZE; i++)
		xil_printf("Ciphertext message value: %x\r\n", ciphertext[i]);

	for (i = 0; i < FRAME_SIZE; i++)
		decrypted[i] = 0;

	/*Decryption process*/
	status = crypto_box_open(decrypted, ciphertext,
			(unsigned long long) FRAME_SIZE, nonce, apk, bsk);

	if (status != 0) {
	xil_printf("ERROR: The frame has not been correctly decrypted!!\n");
		return 1;
	}

	for (j = 0; j < crypto_box_ZEROBYTES; j++) {
		xil_printf("Decrypted message value: %x\r\n", decrypted[j]);
		if (decrypted[j] != 0) {
		xil_printf("ERROR: the first 32 bytes of the decrypted message are
not 0 (incorrect padding)!!\r\n");
	return 1;
		}
	}
	for (i = 32; i < FRAME_SIZE; i++)
		xil_printf("Decrypted message value: %x\r\n", decrypted[i]);

	xil_printf("Cryptographic process correctly finished!!\r\n");

	return 0;
}
```

Para realizar el perfilado se ha empleado Valgrind [75], un conjunto de herramientas *open source* que permiten depurar tanto el rendimiento de un programa como su consumo de memoria.

Del *software* que comprende se ha hecho uso de Callgrind, que permite obtener el perfil de ejecución de la aplicación y determinar la carga computacional de cada una de las funciones que la integran. La información que proporciona a su salida es un fichero *callgrind.out.<pid>* (***pid*** es el identificador de la aplicación perfilada) que incluye, de forma resumida, el número de instrucciones ejecutadas en cada llamada a una función concreta, así como las relaciones existentes entre funciones (llamadas entre ellas); hay que indicar que los costes computacionales de una función incluyen tanto los derivados de su ejecución como el de aquellas funciones que contiene, si se da el caso [75]. Para lanzar la herramienta no es necesario recompilar previamente la aplicación, sino ejecutar en un terminal la siguiente orden:

```
valgrind --tool=callgrind [callgrind options] your-program [program options]
```

Una vez obtenido el fichero de salida, para visualizar de forma gráfica el grafo de llamadas generado es necesario hacer uso de la herramienta KCachegrind, que hace la información generada más legible al diseñador con el objetivo de su estudio. Esta herramienta toma como entrada el fichero de salida generado por Callgrind y realiza la representación de un árbol jerárquico de los costes derivados de la llamada a cada una de las funciones que incluye la aplicación [75]. Para nuestra aplicación en concreto, se obtuvo el árbol de funciones que se muestra en la Figura 32.

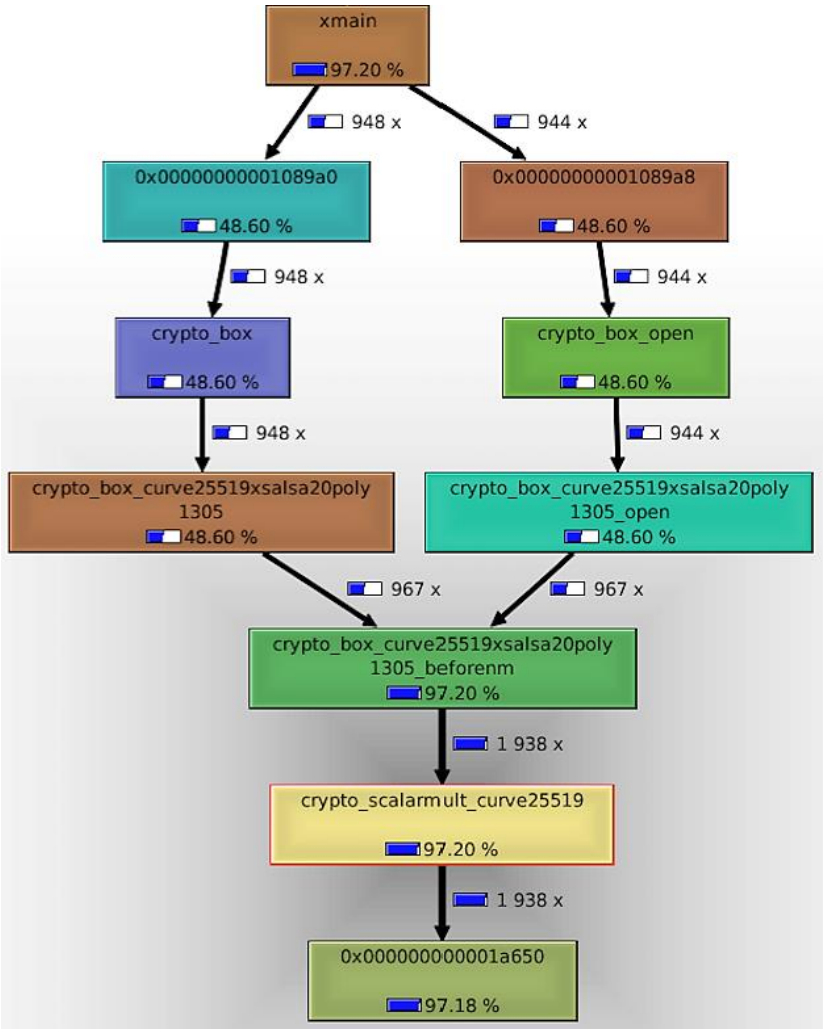



**Figura 32. Grafo de llamadas de la aplicación TweetNaCl.**

Tal como se desprende del gráfico de la figura, la función que mayor carga computacional presenta es la multiplicación escalar que tiene lugar dentro del algoritmo de curvas elípticas curve25519 y a la que se llama durante la generación de las claves públicas y en las etapas de cifrado y descifrado de la información. Por tanto, se presenta como aquella función que precisa con mayor urgencia su aceleración en *hardware* para no comprometer las prestaciones del sistema.

### 4.2.2. Modelado sobre SDSoC

Una vez sabemos que la multiplicación escalar es la función de la librería que precisa de una mayor carga computacional, es turno de pasar a SDSoC para realizar la partición *hardware/software*. De forma previa, hay que resaltar que SDSoC cuenta con sus propias herramientas de perfilado, tanto empleando el tradicional *gprof* como el *TCF profiler*. Sin

embargo, el entorno de perfilado aún no se encuentra lo suficientemente optimizado dentro de la herramienta y la información que proporciona es algo confusa (carga computacional de las instrucciones a nivel de ensamblador), por lo que se ha partido de la información proporcionada por la herramienta Callgrind.

El primer paso en SDSoC es la configuración del proyecto, debiendo determinar con qué tipo de aplicación vamos a trabajar, qué tareas se desea que realice el *software* y qué funciones de la librería son las que se desea trasladar a la lógica programable del dispositivo Zynq, así como la función principal de entrada al programa (la función *main*). En nuestro caso, aunque en un principio se pensaba modelar una aplicación *standalone* (sin sistema operativo), el compilador de SDSoC presentaba errores al no contar con los ficheros necesarios para implementar aplicaciones de este tipo sobre la arquitectura de 32 bits que presentan los núcleos ARM que incluye el dispositivo Zynq; por tanto, se tuvo que optar por una aplicación que incluyera, además de la librería criptográfica, un PetaLinux, lo cual facilita el control de errores de la aplicación al poder hacer uso de las librerías estándar de C.

Por otro lado, como el objetivo fundamental era analizar el rendimiento de la aplicación una vez movida la función que realiza la multiplicación escalar a *hardware*, se ha activado la opción *Estimate Performance*, junto con la generación del *bitstream* final del sistema resultante y de los ficheros necesarios para "bootear" el sistema desde una tarjeta SD. De esta forma, el panel de configuración queda tal como se muestra en la Figura 33.

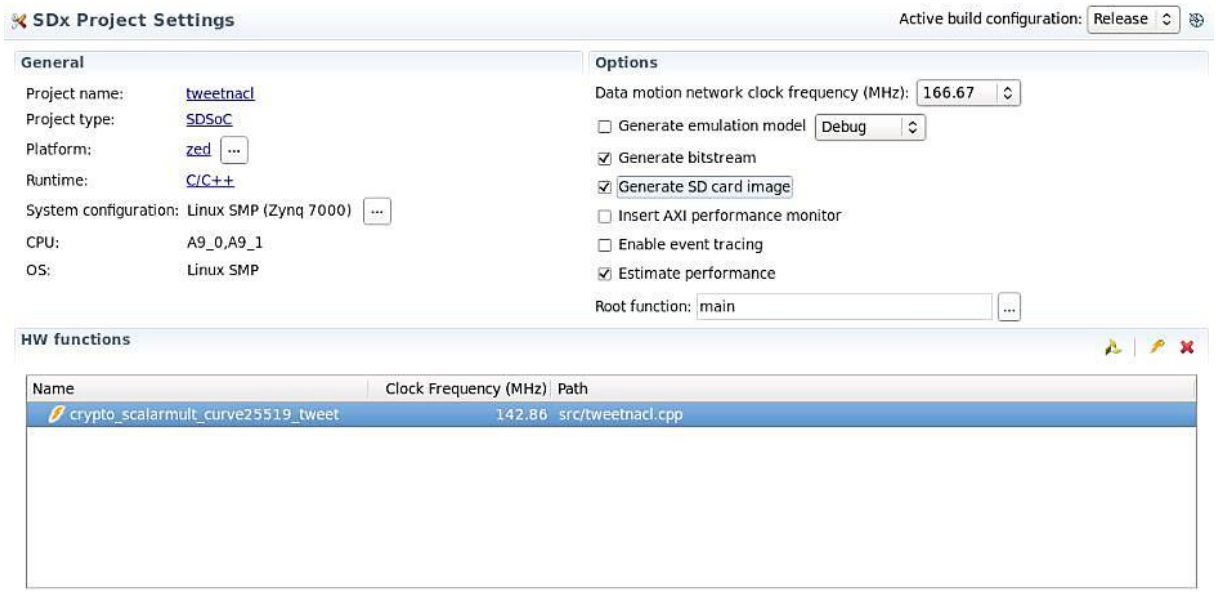



**Figura 33. Panel de configuración en SDSoC.**

Antes de comenzar con la compilación, se hizo un análisis de la aplicación y se incluyeron en el fichero aquellas directivas de optimización que podían hacer que las prestaciones de la librería aumentaran; en este caso, se han empleado las directivas de Vivado HLS, las cuales son totalmente compatibles con SDSoC, UNROLL para el desenrollado de bucles sin dependencia de datos en las instrucciones de su cuerpo y PIPELINE en el cuerpo de la multiplicación escalar, ya que esta función es propicia a comenzar la siguiente instrucción sin haber concluido la anterior, al no haber una dependencia de operaciones de lectura/escritura entre ellas.

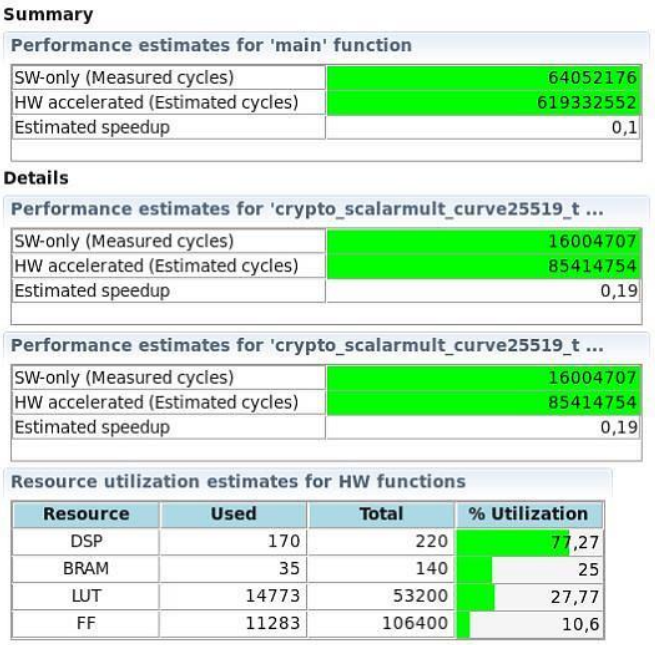
La compilación de la aplicación realiza diferentes tareas, incluyendo la migración de la multiplicación escalar a *hardware* y la generación de toda la infraestructura de comunicación con el resto de la librería que se ha mantenido corriendo en *software* sobre uno de los núcleos ARM de los que dispone el dispositivo Zynq. Al final de este proceso, la herramienta devuelve un *report* sobre las características de las interfaces de comunicación generadas en el bloque acelerador, así como el tiempo de transferencia de datos, en ciclos de CPU, entre la parte *hardware* y el resto del *software* que comprende el sistema, tal como se recoge en la Tabla 7.

**Tabla 7. Interfaces de comunicación del bloque acelerador.**

| Acelerador | Argumento | Sentido | Tamaño(B) | Conexión | Tiempo de transferencia (ciclos de CPU) |
|---|---|---|---|---|---|
| **Crypto_scalarmult** | q | OUT | 32 | ps7_M_AXI_GP0: AXIFIFO | 62 |
| | n | IN | 32 | ps7_M_AXI_GP0: AXIFIFO | 62 |
| | p | IN | 32 | ps7_M_AXI_GP0: AXIFIFO | 62 |
| | return | OUT | 4 | ps7_M_AXI_GP0: AXILITE:0x20C | 13 |

Finalmente, se analiza el rendimiento de la aplicación que, además de proporcionar datos sobre la implementación en *hardware*, permite ejecutar el programa en *software* y realizar una comparativa con los resultados obtenidos. Para la librería TweetNaCl, moviendo

a *hardware* únicamente la función de la multiplicación escalar, se obtienen los resultados mostrados en la Figura 34:



**Summary**

**Performance estimates for 'main' function**

| | |
|---|---|
| SW-only (Measured cycles) | 64052176 |
| HW accelerated (Estimated cycles) | 619332552 |
| Estimated speedup | 0,1 |

**Details**

**Performance estimates for 'crypto_scalarmult_curve25519_t ...**

| | |
|---|---|
| SW-only (Measured cycles) | 16004707 |
| HW accelerated (Estimated cycles) | 85414754 |
| Estimated speedup | 0,19 |

**Performance estimates for 'crypto_scalarmult_curve25519_t ...**

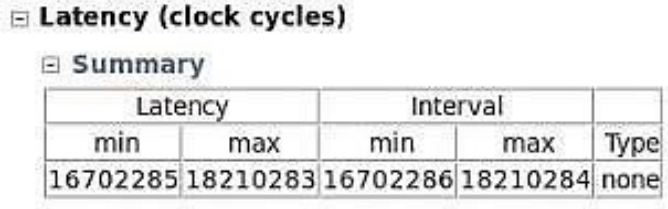
| | |
|---|---|
| SW-only (Measured cycles) | 16004707 |
| HW accelerated (Estimated cycles) | 85414754 |
| Estimated speedup | 0,19 |

**Resource utilization estimates for HW functions**

| Resource | Used | Total | % Utilization |
|---|---|---|---|
| DSP | 170 | 220 | 77,27 |
| BRAM | 35 | 140 | 25 |
| LUT | 14773 | 53200 | 27,77 |
| FF | 11283 | 106400 | 10,6 |

**Figura 34. Estimación de prestaciones de TweetNaCl en SDSoC.**

Tal como se desprende de la información obtenida, el sistema no presenta mejoras en la aceleración, con un *speedup* inferior a 1, lo que indica que la solución adoptada no mejora a la versión puramente *software*, que es lo que cabría esperar al llevar parte de la funcionalidad de la librería a un dispositivo *hardware* acelerador. Sin embargo, los resultados proporcionados son algo confusos ya que, al lanzar Vivado HLS desde el propio entorno SDSoC, opción que se incluye para permitir la optimización por parte del diseñador en caso de que lo considere oportuno, al observar en el informe de síntesis de alto nivel de la función de la multiplicación escalar los resultados de latencia obtenidos, se aprecia que son del orden de casi 70 millones de ciclos menor (Figura 35), lo que daría un *speedup* aproximado de 1, lo cual seguiría conduciendo a descartar esta alternativa, ya que indicaría que no existe ningún tipo de mejora respecto a la implementación inicial.



**Latency (clock cycles)**

**Summary**

| Latency | | Interval | | |
|---|---|---|---|---|
| min | max | min | max | Type |
| 16702285 | 18210283 | 16702286 | 18210284 | none |

**Figura 35. Latencia de la multiplicación escalar en Vivado HLS.**

Esta situación refleja que la toma de decisiones que realiza la herramienta a la hora de realizar la partición *hardware/software* del sistema no es la apropiada, creando cuellos de botella especialmente en las transferencias de datos entre las particiones generadas y en los accesos a memoria por parte de la microarquitectura *hardware*. Por tanto, se ha concluido que SDSoC puede ser útil para diseños de reducida complejidad o a la hora de obtener una primera aproximación a una arquitectura *hardware* de un modelo realizado, pero no resulta eficiente para la implementación de sistemas completos, en los que es clave tener una visión arquitectural de las partes que deben constituir el citado sistema.

Por las razones anteriormente expuestas, se ha descartado el empleo de la librería TweetNaCl en nuestro sistema, al no proporcionar ninguna ventaja acelerar aquella funcionalidad con un mayor coste computacional frente a la solución puramente *software*.

## 4.3. Análisis del bloque de cifrado Simon

Con objeto de dotar al sistema de un bloque de cifrado ligero, a la vista de los problemas de la implementación de la librería TweetNaCl, se ha optado por modelar en C/C++, haciendo uso de la herramienta de Vivado HLS, el bloque de cifrado Simon de clave simétrica [61], adaptando su especificación para que sea capaz de manejar paquetes Ethernet ya que, en su versión estándar, se limita a cifrar dos palabras de 32 bits o de 64 bits, mediante el empleo de operaciones binarias y de rotación de datos entre ellas.

Para este sistema, se ha optado por modelar la alternativa Simon de 64 bits de palabra de entrada (dividida en dos fragmentos de 32 bits para la aplicación del algoritmo) para que exista compatibilidad con el ancho del campo de datos definido para la etapa de filtrado, haciendo uso de una clave con una longitud de 42 palabras de 32 bits y realizando un total de 42 rondas, lo recomendable para generar un algoritmo robusto frente a los ataques más comunes que tienen lugar en una arquitectura IoT.

Asimismo, hay que indicar que se ha decidido modelar de forma independiente tanto la funcionalidad de cifrado como de descifrado. Con esta partición, lo que se pretende es dotar a la aplicación de un alto grado de modularidad, insertando únicamente en el sistema aquella operación, mediante el bloque IP oportuno, que sea requerida (cifrado, descifrado o

ambas). Hay que resaltar que esta separación no provoca un aumento de la latencia del sistema.

Finalmente, indicar que el código fuente de partida ha sido tomado del proyecto BLOC de la *French National Research Agency*, el cual está disponible con licencia *open source* para su utilización [76].

### 4.3.1. Modelado de la funcionalidad

El algoritmo comienza, tanto en el bloque IP de cifrado como en el de descifrado, con una etapa de expansión de la clave única, de forma que como argumento de entrada se define el tamaño que debe tener la clave (en nuestro caso, un array de 42 palabras de 32 bits) y se definen de forma pseudo-aleatoria sus 3 dos primeros valores. En este punto, el algoritmo llama a la función *KeyExpansion* que, combinando los valores proporcionados y los generados a partir de estos de forma iterativa, mediante la aplicación de operaciones binarias XOR y de desplazamientos a la derecha, permite obtener las 32 palabras de que consta la clave simétrica que regirá el funcionamiento del bloque de cifrado.

De forma resumida, esta etapa se basa en el desplazamiento a la derecha por 3 y una operación XOR entre este desplazamiento y el desplazamiento a la derecha una unidad de la palabra de la clave única con mayor índice y, posteriormente, se aplica la operación XOR con la palabra de 32 bits de menor índice y una constante de rotación $z$ previamente definida. Al final de cada ronda, la nueva palabra de 32 bits generada se escribe en el índice inmediatamente superior a los empleados para la expansión y desplazan todas las palabras una posición a la derecha, de forma que aquella con menor índice ya no participará en las operaciones de la siguiente ronda. Para su correcta ejecución, es necesaria la realización de 38 rondas.

```
void KeyExpansion(u32 key[KEY_SIZE]) {
	status = 2;
	i = 0;
	text_tmp = 0;
	for (i = 3; i < 42; i++) {
		text_tmp = ROTATE_RIGHT_32(key[i - 1], 3);
		text_tmp = text_tmp ^ ROTATE_RIGHT_32(text_tmp, 1);
		key[i] = ~key[i - 3] ^ text_tmp ^ z[i - 3] ^ 3;
	}
	printf("Key expansion finished!!\r\n\n");
}
```

Por otra parte, las funciones de cifrado y descifrado de ambos bloques presentan una implementación similar, ya que se trata de operaciones totalmente reversibles a las que, aplicando la misma clave única, permiten obtener el mensaje original. Como argumentos de entrada, precisan tanto de la clave única como del paquete de entrada sobre el que trabajar (original o cifrado, en función de la operación seleccionada) y del *buffer* de salida donde se almacena el resultado de la operación.

Indicar que los datos de entrada que se le suministran a las operaciones principales de los bloques IP no deben incluir en ningún caso información de la cabecera de los paquetes. Esta información no puede ser cifrada con el fin de no modificar las rutas de direccionamiento y que el paquete en cuestión siempre llegue al destino final. Por tanto, la cabecera será extraída al comienzo de la operación y se ensamblará al comienzo del paquete antes de devolverlo por la interfaz AXI4-Stream de salida.

Cuando se cuenta con estos parámetros, se realiza una copia del mensaje en un *buffer* local a la función sobre el cual se va a operar y, mientras la entrada aplicada tenga datos disponibles, se realizan las operaciones binarias descritas en el apartado de Bloques de cifrado compactos. Concretamente, en cada ronda se realiza un desplazamiento a la izquierda de 1, 2 y 8 posiciones y una operación binaria AND entre los desplazamientos de 1 y 8 posiciones de una de las palabras iniciales (la de menor índice para el cifrado, la de mayor índice para el descifrado) para, posteriormente, realizar una operación XOR con la otra palabra no empleada y la palabra de 32 bits de la clave única correspondiente a esa ronda en concreto.

Una vez que se han realizado todas las rondas para todos los pares de palabras de 32 bits de que consta el mensaje proporcionado, se copia en el *buffer* de salida de la función la información disponible en el *buffer* local, siendo este contenido el texto cifrado en el caso de la operación de encriptación y el mensaje original en el caso de la función de descifrado. En la Figura 36 se resume el flujo de trabajo de los bloques IP modelados para formar parte de la etapa de cifrado del sistema.

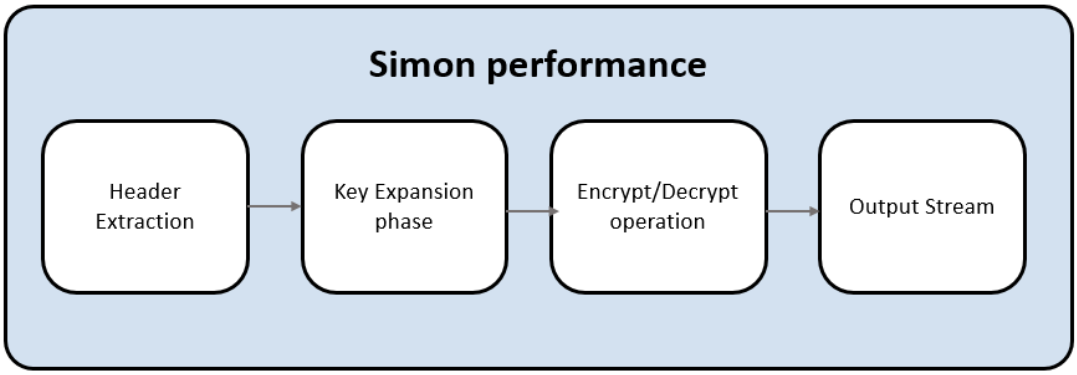



**Figura 36. Flujo de trabajo de los bloques Simon.**

A continuación, se muestra el modelado para la función de cifrado siendo, tal como se ha mencionado, el de la función de descifrado idéntico, pero comenzando las rondas con el último valor de la clave única.

```
ap_int<3> Encrypt(u32 key[KEY_SIZE], ap_uint<8> length, u32* text, u32* crypt) {

	short size = length;
	status = 2;
	text_tmp = 0;

	for (i = 0; i < PAYLOAD_SIZE; i++)
#pragma HLS UNROLL
		text_reg[i] = 0;

	for (k = 0; k < size; k++) {
		text_reg[k] = text[k];
		printf("Text value nº%u: %u\n\n\n", k + 1, (u32) text_reg[k]);
	}

	for (k = 0; k < size - 1; k++) {
		for (i = 0; i < 42; i++) {
			text_tmp = text_reg[0 + (2 * k)];
			text_reg[0 + (2 * k)] = text_reg[1 + (2 * k)]
					^ ((ROTATE_LEFT_32(text_reg[0 + (2 * k)], 1))
						& (ROTATE_LEFT_32(text_reg[0 + (2 * k)],
8)))
					^ (ROTATE_LEFT_32(text_reg[0 + (2 * k)], 2)) ^
key[i];
			text_reg[1 + (2 * k)] = text_tmp;
		}
		printf("Crypted value nº%u: %u\n\n\n", k + 1, (u32) text_reg[k]);
	}

	for (k = 0; k < size; k++) {
		crypt[k] = text_reg[k];
		printf("Ciphertext value nº%u: %u\n\n\n", k + 1, (u32) crypt[k]);
	}
	status = 0;
	return status;
}
```

### 4.3.2. Definición de la función *top* y de las interfaces de comunicación

Una vez modelada y verificada la funcionalidad de los bloques de cifrado y descifrado en C/C++, el siguiente paso es la definición de la función *top* que controla el correcto funcionamiento del bloque IP. Asimismo, se hace necesario definir las interfaces de comunicación para la realización de transacciones con los filtros de autenticación y con el resto de bloques IP que compondrán el sistema final.

Para alimentar la funcionalidad de estos bloques IP, es necesario registrar con anterioridad los datos recibidos a través de la interfaz de entrada hasta que se recibe una señal de fin de paquete (TLAST = 1).

Asimismo, se ha realizado la extracción del campo de la cabecera *Payload Length*, con el fin de indicarle al bloque IP en cuestión la cantidad de palabras de 32 bits existentes en el *payload* del paquete recibido y, por tanto, el número de iteraciones que debe realizar el algoritmo para completar el proceso completo de cifrado o descifrado.

Una vez devuelta una respuesta satisfactoria sobre su operación, el siguiente paso consiste en poner los datos generados en la interfaz de salida, devolviendo el mensaje resultante hacia el bloque de procesamiento, para que realice con él la acción oportuna en función de la respuesta dada por la etapa de filtrado. En este punto, es importante gestionar de forma correcta todas las señales de la interfaz AXI4-Stream, incluyendo tanto el *flag* de fin de paquete (TLAST = 1) como la señal TKEEP, que debe tener el valor 0xF para indicar que todos los datos disponibles en el campo TDATA de la interfaz son válidos durante la transacción en curso. Finalmente, se incluye una serie de etapas de control de errores, mediante la evaluación de la variable `status` y diversos mensajes de estado que se muestran por consola.

A continuación se muestra un extracto de la función *top* utilizada, incluyendo la definición de las interfaces, los parámetros de entrada y salida y otros aspectos de su funcionalidad y control de los protocolos.

```
ap_int<3> simon_block_cipher(u32 key[KEY_SIZE], ap_axis<32, 1, 1, 1> * data_in,
            ap_axis<32, 1, 1, 1> * data_out) {
#pragma HLS INTERFACE s_axilite port=return bundle=axi_lite
#pragma HLS INTERFACE axis register both depth=400 port=data_in name=frame_in
#pragma HLS INTERFACE axis register both depth=400 port=data_out name=frame_out
#pragma HLS INTERFACE s_axilite port=key bundle=axi_lite

      ap_uint<9> word_id;
      ap_uint<9> num_iterations;
      ap_uint<9> max_iterations;
      ap_uint<1> in_last;
      ap_uint<16> payload_len;
      ap_uint<32> header_reg[HEADER_SIZE];
      u32 data_reg[PAYLOAD_SIZE];
      u32 data_exit[PAYLOAD_SIZE];

      u32 key_tmp[KEY_SIZE];

      status = 2;
      word_id = 0;
      num_iterations = 0;
      max_iterations = 0;
      payload_len = 0;
      in_last = 0;
      k = 0;

      for (k = 0; k < PAYLOAD_SIZE; k++)
            data_reg[k] = 0, data_exit[k] = 0;

      for (k = 0; k < HEADER_SIZE; k++)
            header_reg[k] = 0;

      for (word_id = 0; word_id < HEADER_SIZE; word_id++) {
            header_reg[word_id] = data_in->data;
            data_in++;
            printf("Header value: %u \r\n\n", (u32) header_reg[word_id]);
      }

      for (word_id = 0; word_id < PAYLOAD_SIZE; word_id++) {
            data_reg[word_id] = data_in->data;
            in_last = data_in->last;
            data_in++;
            if (in_last == 1)
                  break;

      }

      payload_len = header_reg[4].range(15, 0);
      payload_len = payload_len - CHECKSUM_SIZE;

      if ((payload_len % WORD_FORMAT) != 0) {
            num_iterations = (payload_len / WORD_FORMAT) + 1;
      } else {
            num_iterations = (payload_len / WORD_FORMAT);
      }

      max_iterations = num_iterations;

      for (k = 0; k < KEY_SIZE; k++) {
```

```
		key_tmp[k] = 0;
		key_tmp[k] = key[k];
	}

	KeyExpansion(key_tmp);

	status = Decrypt(key_tmp, num_iterations, (u32*) data_reg, data_exit);

	if (status) {
		printf("ERROR: decryption process failed!!\n");
		return 1;
	}

	for (word_id = 0; word_id < HEADER_SIZE; word_id++) {
		data_out->data = header_reg[word_id];
		data_out->keep = 255;
		data_out->strb = 255;
		data_out->user = 0;
		data_out->last = 0;
		data_out->id = 0;
		data_out->dest = 0;
		data_out++;

		printf("Header value: %u \r\n", (u32) header_reg[word_id]);
	}

	for (k = 0; k < max_iterations; k++) {
		data_out->data = data_exit[k];
		data_out->keep = 255;
		data_out->strb = 255;
		data_out->user = 0;
		data_out->last = 0;
		data_out->id = 0;
		data_out->dest = 0;
		data_out++;

		printf("Decrypted value: %u \r\n", (u32) data_exit[k]);
	}

	data_out->data = 0;
	data_out->keep = 1;
	data_out->strb = 1;
	data_out->user = 0;
	data_out->last = 1;
	data_out->id = 0;
	data_out->dest = 0;
	data_out++;

	printf("Out axi_stream signal correctly copied\n");
	return status;
}
```

En referencia a las interfaces de comunicación, contamos con 2 interfaces de entrada y 2 de salida:

- *Key*. Se corresponde con las 42 palabras de 32 bits que forman la clave única.

- `Data_in` y `Data_out`. Paquetes de datos a cifrar/descifrar por el módulo y mensaje resultante, respectivamente.
- `Status`. Indica si la operación aplicada se ha realizado de forma correcta.

En cuanto al tipo de interfaz AXI4 definido en cada caso se ha optado, al igual que en los módulos de autenticación, por definir la interfaz de entrada correspondiente a la clave única como AXI4-Lite esclava que actúa como puerto de configuración. Asimismo, la interfaz de salida `status` también se define como AXI4-Lite, para retornar al PS el resultado derivado de la ejecución de las operaciones de cifrado o descifrado, en función del bloque IP empleado. También en este caso, todas las interfaces AXI4-Lite definidas se incluyen en un mismo bus.

Por otro lado, tanto la interfaz de entrada `data_in` como la de salida `data_out` se han definido como AXI4-Stream, para permitir una arquitectura tipo *dataflow* en la que el tráfico de información es constante en el sistema en caso de que haya datos disponibles. Como en el caso de los filtros basados en CBFs, la profundidad de ambas interfaces presenta un valor de 400, para poder hacer frente al tamaño máximo de un paquete IEEE 802.3.

En la Figura 37 se puede observar el diagrama de entrada/salida del bloque de cifrado Simon implementado:

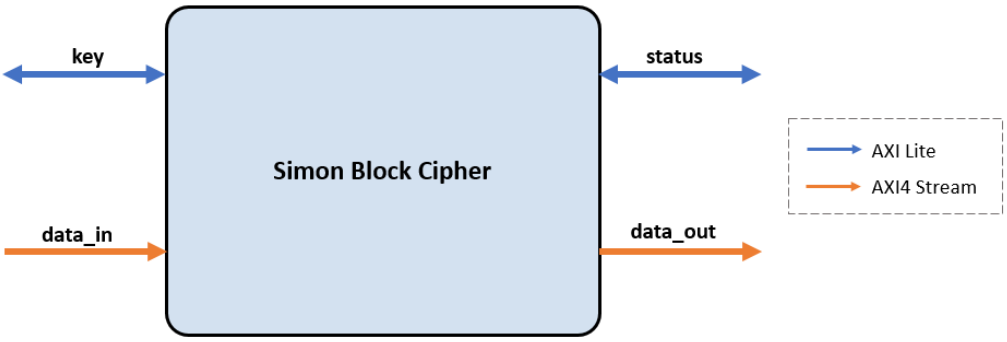



**Figura 37. Diagramas de E/S de los bloques Simon.**

### 4.3.3. Verificación y síntesis de alto nivel

Definida la función *top*, hay que asegurarse de que el bloque IP realiza las operaciones asociadas de la forma esperada. Con este propósito, se ha diseñado el *testbench* correspondiente para verificar las distintas situaciones que pueden darse en el bloque IP cuando se encuentra en estado de ejecución. De esta forma, se evita realizar la

detección de errores *a posteriori* en otras etapas del flujo de diseño, en donde la cantidad de módulos que integran la plataforma se incrementa y resulta más difícil depurar el sistema.

Para esta etapa de cifrado, se ha comprobado que el bloque IP correspondiente en cada caso realiza de forma correcta la operación de cifrado/descifrado asociada, así como la posterior escritura de los resultados obtenidos en la señal AXI4-Stream de salida. Adicionalmente, igual que se hizo durante el modelado de la función *top*, se han incluido mecanismos de detección de errores, basados en el estado de la variable de `status` y en mensajes impresos en consola, así como una función denominada *configPayload*, a través de la que se le proporcionan a las señales AXI4-Stream de test los valores oportunos para todas sus señales. Finalmente, se comprueba que el paquete obtenido a la salida del bloque de descifrado coincide en todas sus palabras de 32 bits con el mensaje original aplicado como entrada al bloque de cifrado.

En la siguiente captura se muestra la forma que presenta el *testbench* para el caso del bloque de descifrado:

```
int main() {

	ap_int<3> status_t = 2;

	//Key definition
	u32 key_t[KEY_SIZE];

	for (int i = 0; i < KEY_SIZE; i++)
		key_t[i] = 0;

	key_t[0] = 0x03020100;
	key_t[1] = 0x0b0a0908;
	key_t[2] = 0x13121110;

	for (int i = 0; i < PAYLOAD_SIZE; i++)
		frame[i].data = 0,ciphertext_in[i].data = 0, payload[i].data = 0;

	printf("\n Test running...\n");

	configPayload(frame);

	printf("\n Decryption process starts!!\r\n\n");
	configPayload1(ciphertext_in);
	status_t = simon_block_cipher(key_t, ciphertext_in, payload);
	printf("\n Status value: %u\r\n\n", (unsigned int) status_t);

	if (status_t != 0) {
		printf("ERROR: decryption process failed!!\n");
		return 1;
	}
```

```
	//Verification phase
	for (int i = 0; i < PAYLOAD_SIZE; i++) {

		if (payload[i].last == 1)
			break;

		if ((u32) frame[i].data != (u32) payload[i].data) {
			printf("ERROR: Original value is not equal that the final
value!!\n");
			return 1;
		}
		printf("Original value %u: %llu \r\n"
				"Final value %u: %llu \r\n\n\n", i, (u32) frame[i].data,
i,
				(u32) payload[i].data);
	}

	printf("\n Decryption process finished successfully!!\r\n");

	printf("\n Status value: %u\r\n\n", (unsigned int) status_t);

	printf("\n ---Exiting main()---\r\n");

	return 0;
}
```

Tras haber realizado la verificación de ambos bloques IP mediante el *testbench* anteriormente expuesto, es turno ahora de centrarse en el proceso de síntesis de alto nivel.

Mediante el análisis previo realizado, se han detectado partes del código propicias a ser optimizadas, como son diversos bucles de *reset* (escritura de un valor 0 en todas las posiciones de un *array* empleado como *buffer*) que pueden ser completamente desenrrollados haciendo uso de la directiva UNROLL. Además, se planteó aplicar a las rondas de cifrado/descifrado la directiva PIPELINE, que permite comenzar la operación posterior sin haber concluido completamente la actual. Sin embargo, esta última operación se observó posteriormente que no era aplicable al diseño, debido a que en dicho bucle existían operaciones de lectura y escritura sobre una misma variable (en este caso, la variable *text_tmp*).

En la Tabla 8 y la Tabla 9 se pueden observar los resultados obtenidos para latencia y recursos lógicos consumidos, respectivamente, durante el proceso de síntesis de alto nivel de la etapa de cifrado del sistema. Además, se ha incluido en este caso los resultados obtenidos para el dispositivo Zynq UltraScale+, con el fin de obtener una visión global de las prestaciones del bloque IP en función del dispositivo sobre el que finalmente se implemente.

**Tabla 8. Resultados de *timing* en síntesis de los bloques Simon.**

| | Zynq | | UltraScale+ | |
|---|---|---|---|---|
| | **Encrypt** | **Decrypt** | **Encrypt** | **Decrypt** |
| **Periodo de reloj objetivo** | 6,50 ns | 6,50 ns | 6,50 ns | 6,50 ns |
| **Periodo de reloj estimado** | 5,58 ns | 5,32 ns | 3,58 ns | 3,58 ns |
| **Incertidumbre** | 0,81 ns | 0,81 ns | 0,81 ns | 0,81 ns |

**Tabla 9. Consumo de recursos en síntesis para los bloques Simon.**

| | Zynq | | UltraScale+ | |
|---|---|---|---|---|
| | **Encrypt** | **Decrypt** | **Encrypt** | **Decrypt** |
| **BRAM_18K** | 8/280 (2%) | 2/280 (<1%) | 8/1.824 (<1%) | 8/1.824 (<1%) |
| **DSP48E** | 0/220 (0%) | 0/220 (0%) | 0/2.520 (0%) | 0/2.520 (0%) |
| **FF** | 1.004/106.400 (<1%) | 430/106.400 (<1%) | 815/548.160 (<1%) | 815/548.160 (<1%) |
| **LUT** | 4.143/53.200 (7%) | 3.013/53.200 (5%) | 4.119/274.080 (1%) | 4.134/274.080 (1%) |

Como se observa en los resultados de la Tabla 8, el *timing* objetivo es, en ambos casos, algo superior a la suma del tiempo estimado más la incertidumbre definida. De esta forma, podemos concluir que la estimación previa realizada ha sido correcta. Asimismo, con la información obtenida se resaltan las altas prestaciones que presenta el dispositivo Zynq UltraScale+ por lo que, para futuros proyectos donde se requieran bloques IP que realicen operaciones aritméticas sumamente complejas o en repetidas ocasiones, se recomienda el empleo de este dispositivo.

### 4.3.4. Cosimulación y exportación del bloque IP

Finalizada la etapa de síntesis de alto nivel, el siguiente paso consiste en verificar, mediante el *testbench* en lenguaje HDL equivalente al modelado en C/C++, que la descripción RTL generada presenta el mismo comportamiento que el algoritmo implementado en lenguaje de alto nivel.

Ejecutando este proceso se obtienen, tras haber seleccionado nuevamente Verilog como lenguaje HDL, los resultados reflejados en la Tabla 10, en los que se puede apreciar que, además de pasar satisfactoriamente el test generado, el bloque IP de descifrado toma 168 ciclos más de media para finalizar su ejecución en el peor de los casos que el bloque de cifrado. De este modo, teniendo en cuenta los tiempos de reloj estimados durante la síntesis de alto nivel, obtenemos una latencia de **10,864 µs** para el bloque de cifrado y de **11,252 µs** para el bloque de descifrado.

Asimismo, hay que resaltar que, para ambos bloques, se dispone de un intervalo de 9 ciclos, siendo esta la barrera en la cual una operación posterior puede iniciarse sin haber finalizado la actual. De esta forma, este bloque IP también explota las características de paralelismo que presenta un *hardware* con naturaleza aceleradora como es la FPGA.

**Tabla 10. Resultados de cosimulación para los bloques Simon.**

| | | | Latencia (ciclos) | | | Intervalo | | |
|---|---|---|---|---|---|---|---|---|
| **IP** | RTL | Status | Mínimo | Media | Máximo | Mínimo | Media | Máximo |
| **Encrypt** | Verilog | Pass | 1.881 | 1.947 | 2.009 | 1.872 | 1.938 | 2.000 |
| **Decrypt** | Verilog | Pass | 2.084 | 2.115 | 2.179 | 2.075 | 2.106 | 2.170 |

Llegados a este punto, es necesario resaltar que la diferencia en la cantidad de ciclos que toman estos bloques para finalizar todas las instrucciones indicadas en comparación con la etapa de filtrado para IoT es tan significativa debido a la complejidad de las operaciones que realizan, así como a la cantidad de veces que son ejecutadas, estando condicionados los bloques Simon por la cantidad de rondas a realizar independientemente de la operación realizada (cifrado o descifrado). Aunque cabría pensar que el número de rondas realizadas es desmedido y podría reducirse, se ha mantenido esta cantidad ya que es la que define el estándar para manejar palabras de entrada de 64 bits y comportarse de forma robusta frente a los ataques más comunes que afectan a los dispositivos integrados en una arquitectura IoT.

Lanzando el visualizador de formas de onda que integra Vivado HLS y que se encuentra disponible tras finalizar correctamente la etapa de cosimulación, podemos verificar que el bloque IP se comporta en sus diferentes etapas como cabía esperar. En la

Figura 38 y la Figura 39 se muestra un caso de cifrado, reflejándose los datos de entrada y el texto cifrado resultante, respectivamente.

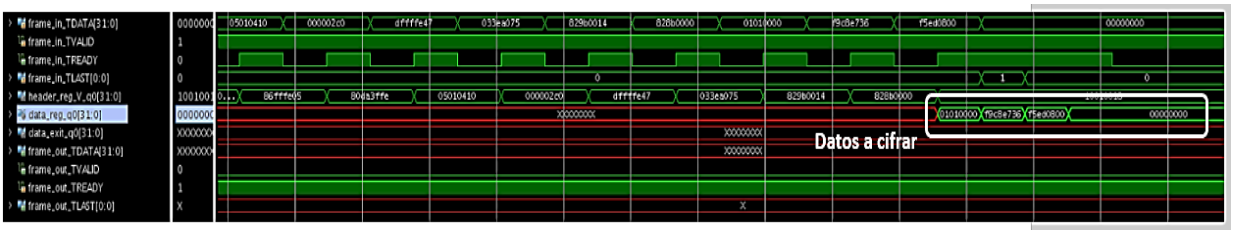



**Figura 38. Datos de entrada al bloque de cifrado.**

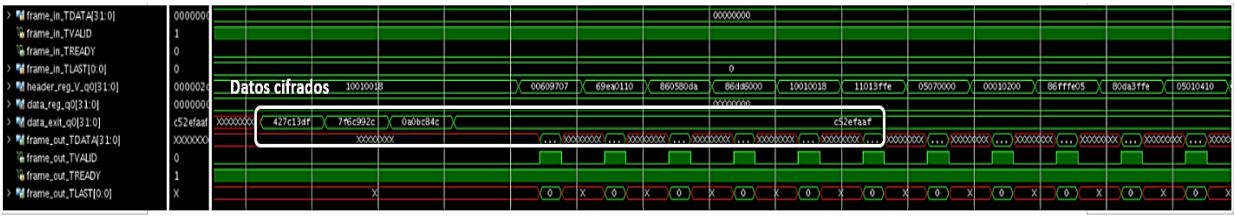



**Figura 39. Salida del bloque de cifrado.**

En la Figura 40 y la Figura 41, por su parte, se puede observar el modo de operación del módulo de descifrado, mostrando el paquete de entrada (el resultante de la operación de cifrado) y el resultado obtenido a la salida, el cual coincide con el mensaje original cifrado en la Figura 38, respectivamente.

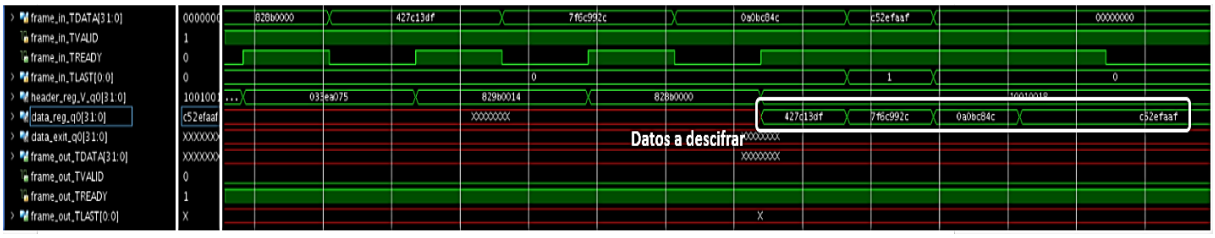



**Figura 40. Datos de entrada al bloque de descifrado.**

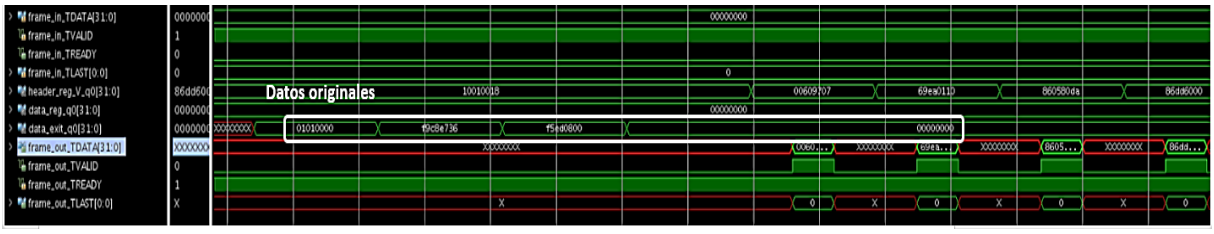



**Figura 41. Salida del bloque de descifrado.**

Una vez que se consideran los resultados de cosimulación como aceptables para cumplir la aplicación para la que han sido concebidos estos bloques IP, es momento de

exportarlos para concluir con el flujo de diseño en alto nivel y que puedan ser integrados con el resto de la plataforma, si se considera necesario, que conformará el sistema final.

En la Tabla 11 se puede comprobar que, con estos ciclos de reloj, los bloques de cifrado y descifrado son capaces de trabajar a una frecuencia de unos 156 MHz y 160 MHz, respectivamente. Además, se observa que los resultados finales son peores que los obtenidos durante la etapa de síntesis de alto nivel, debido a que es durante la exportación cuando se tienen en cuenta las restricciones temporales asociadas a las interfaces. Hay que indicar que, a pesar de que los ciclos de reloj obtenidos son similares, el bloque de cifrado sigue siendo ligeramente más lento que el descifrado.

Por otro lado, en cuanto a los recursos lógicos se refiere, en la Tabla 12 se aprecia también en este caso, igual que ocurría en el proceso de diseño de los filtros de autenticación, un descenso significativo respecto a la estimación realizada en la síntesis de alto nivel, el cual es más considerable para el caso de las LUTs. Asimismo, se observa como para ambos bloques el consumo de recursos final es bastante similar en cada uno de los elementos lógicos disponibles.

**Tabla 11. Resultados de *timing* en exportación de los bloques Simon.**

| | Encrypt | Decrypt |
|---|---|---|
| **Periodo de reloj objetivo** | 6,5 ns | 6,5 ns |
| **Periodo de reloj conseguido** | 6,401 ns | 6,291 ns |

**Tabla 12. Consumo de recursos en exportación de los bloques Simon.**

| Recurso | Encrypt | Decrypt |
|---|---|---|
| **BRAM** | 8 | 8 |
| **DSP** | 0 | 0 |
| **FF** | 875 | 882 |
| **LUT** | 989 | 1043 |

## 4.4. Conclusión

Este capítulo ha servido como estudio de campo de las alternativas criptográficas previamente seleccionadas en el apartado de Alternativas criptográficas como opciones

viables a implementar en un diseño *hardware* como el que tiene como objetivo el presente TFM.

Tras hacer uso de la librería TweetNaCl y realizar una aplicación que haga uso de la funcionalidad que proporciona se ha concluido, haciendo uso de SDSoC, que la aceleración de las funciones más críticas computacionalmente hablando que incluye no proporciona un resultado de *speedup* lo suficientemente satisfactorio como para seguir considerándola como una opción válida. Asimismo, se ha comentado la confusión en los datos de estimación de las prestaciones que proporciona SDSoC, ya que varían desde los obtenidos en el propio entorno con los que refleja el Vivado HLS lanzado desde la propia herramienta.

Por otro lado, se ha modelado mediante una descripción *hardware-friendly* y teniendo como objetivo el cifrado/descifrado de paquetes Ethernet dos bloques IP que aplican el algoritmo Simon, describiendo con detalle su modelado, tanto la funcionalidad como la declaración de las interfaces de comunicación pertinentes. En este caso, los resultados obtenidos sí han sido positivos en términos de latencia y, especialmente, en consumo de recursos lógicos, por lo que se ha decidido mantener esta solución en el esquema del sistema final y será la que conforme la etapa de cifrado modular de la arquitectura segura diseñada para IoT.

# Capítulo 5. Integración hardware/software

## 5.1. Introducción

Una vez que se ha descrito el flujo de diseño de los bloques IP tanto de cifrado como de autenticación y se ha verificado que funcionan de forma correcta, el siguiente paso consiste en abordar su integración con el resto de módulos necesarios para formar el sistema completo, así como definir el *software* empotrado que gobernará su correcto modo de operación en cada momento en función de las necesidades que se presenten.

## 5.2. Arquitectura de la plataforma

Tal como se ha mencionado anteriormente en el apartado de Entorno de diseño, el modelado de un sistema completo en Vivado utiliza una metodología basada en plataformas. De este modo, el sistema estará compuesto por distintos bloques IP, estableciendo entre ellos las comunicaciones oportunas, mediante buses e interfaces AMBA AXI4, para que cumpla con la aplicación objeto de diseño.

Además de los bloques *custom* diseñados empleando la metodología de síntesis de alto nivel (bloques de filtrado y cifrado), es necesario incluir módulos adicionales para que el sistema completo presente el comportamiento deseado. A continuación, se explican todos los bloques IP incluidos en la plataforma, resaltando aquellas características relevantes para el diseño realizado:

- *Processing_system7_0.* Se trata del bloque fundamental de toda plataforma, ya que es una instancia del bloque de procesamiento del dispositivo Xilinx Zynq. Su misión principal es regir el correcto funcionamiento del resto del sistema realizando, entre otras tareas, la comunicación con la FPGA y con las interfaces de E/S, el control de la memoria DDR externa al chip, la gestión de *timers* e interrupciones o la ejecución del *software* empotrado en los núcleos de propósito general que incluye.
- *Rst_processing_system7_0_100M.* Gestiona todas las señales de *reset* del resto de bloques IP que conforman la plataforma, asegurándose de este modo que, tras su ejecución, todos sus parámetros presenten valores conocidos y correspondientes a su configuración inicial.
- *Axi_mem_intercon y ps7_0_axi_periph*. Infraestructura de gestión y configuración de los buses AMBA AXI4 presentes en el diseño, estableciendo los enlaces oportunos entre los distintos bloques IP incluidos en la plataforma, para que el sistema cumpla con la aplicación definida. Además, realizan una adaptación de frecuencias de trabajo a ambos lados de su interconexión permitiendo, de este modo, conectar distintos pares maestro-esclavo que trabajen a tasas de datos diferentes e, incluso, que hagan uso de interfaces AMBA AXI4 distintas.
- *Axis_broadcaster_0.* Debido a que el DMA sólo es capaz de realizar transacciones punto a punto, se hace necesario incluir un bloque a su salida que permita alimentar a los tres bloques IP *custom* con los mismos datos. Este es el motivo de la utilización del AXI4-Stream Broadcaster, el cual replica la información que recibe por el DMA en cada una de sus salidas. Aunque en este diseño únicamente es necesario enviar los paquetes de datos a tres bloques IP, el Broadcaster es capaz de actuar sobre 16 interfaces AXI4-Stream al mismo tiempo.
- *Axi_dma_0.* Gestiona las transacciones entre los bloques *custom* y la memoria del sistema, sin necesidad de intervención por parte de la CPU. De esta forma, el PS es capaz de enviar datos a los bloques IP diseñados. El inconveniente que presenta su incorporación en la plataforma es que el PS es incapaz de ejercer un control directo sobre los bloques IP *custom*, siendo su gestión totalmente transparente y viendo únicamente en su entorno de actuación al DMA.

- *Simon_block_cipher_0.* Bloque IP modelado en alto nivel y que tiene como función principal el cifrado y descifrado de la información (en función del sentido de comunicación, se empleará uno u otro según proceda) que maneja el nodo *Fog*, aplicando el algoritmo Simon basado en redes de Feistel. Se compone, además del algoritmo principal y de las señales de control (reloj y *reset*), de un puerto de configuración AXI4-Lite para definir los 3 primeros valores de la clave única para realizar su expansión y que también se emplea para retornar el estado de las operaciones realizadas, así como de una interfaz de entrada y otra de salida AXI4-Stream, a través de las que se recibe el paquete de datos sobre el que actuar (registrado previamente en memoria y accesible a través del DMA) y se devuelve el mensaje equivalente tras aplicar la operación correspondiente, respectivamente.
- *Udp_cbf_0.* Bloque *custom* diseñado siguiendo una metodología de síntesis de alto nivel y que se encarga de parte de la etapa de filtrado del sistema. Mediante la *whitelist* definida previamente realizando la inserción de los puertos UDP a través de los que se considere que el nodo *Fog* puede recibir solicitudes de comunicación entrantes, permite bloquear o autorizar el acceso al mismo. Para ello, se extrae de los paquetes de datos entrantes el campo correspondiente al puerto UDP de destino y se compara su valor con la lista configurada, tomando la decisión oportuna en función de las políticas de seguridad definidas. Cuenta con un puerto de configuración AXI4-Lite para gestionar las listas, permitiendo la inclusión y eliminación de elementos, así como para devolver hacia el PS el resultado de las operaciones realizadas. Además, dispone de una interfaz de entrada AXI4-Stream para recibir los paquetes Ethernet albergados en memoria a través del DMA para su análisis.
- *Ip_cbf_0.* Bloque IP, también modelado en alto nivel, que completa la infraestructura de filtrado del sistema. En este caso, se define una *blacklist* con aquellas direcciones IPv6 correspondientes a subredes o nodos concretos que se consideren potenciales amenazas para el correcto funcionamiento del sistema, bloqueando los intentos de conexiones provenientes desde los orígenes incluidos en dicha lista. De esa forma, se pretende evitar ataques del tipo DDoS, bloqueando conexiones entrantes desde dispositivos conocidos que actúen como *botnets*. Para este sistema, se ha definido como política de acceso el bloqueo de todos aquellos nodos conectados a una subred

definida como posible amenaza para la operación del nodo *Fog*. Al igual que en el caso del filtro para UDP, hay que extraer el campo correspondiente a la subred de origen (no hará falta extraer el nodo, debido a la política de acceso definida) de la cabecera de los paquetes Ethernet entrantes al sistema y comprobar si su valor coincide con alguno de los valores almacenados previamente en la estructura CBF implementada. Como pasaba también con el filtro para UDP, dispone de un puerto de configuración AXI4-Lite tanto para gestionar las listas como para devolver el resultado de las operaciones realizadas, así como de una interfaz de entrada AXI4-Stream para recibir los paquetes Ethernet que serán objeto de análisis.

- *Axis_data_fifo_[0-4].* Para evitar que ambos extremos de una transacción sobre una interfaz AXI4-Stream tengan que acordar de forma previa la velocidad de transferencia de datos, se incorporan este tipo de FIFOs en medio con el fin de que, independientemente de las frecuencias de trabajo a las que se realicen las operaciones de lectura y escritura sobre ellas, el sistema no se bloquee y dé como resultado un cuello de botella.
- *Integrated Logic Analyzer.* Permiten la monitorización y el depurado de aquellas señales incluidas en los buses AMBA AXI4 sobre los que se desea actuar. Su comportamiento es el de un analizador lógico digital, pudiendo ver y analizar el diagrama de señales en aquellas líneas de comunicación de la plataforma que se consideran críticas para el correcto funcionamiento del sistema completo. Al tratarse de bloques de depurado, una vez que se ha comprobado que el sistema se comporta de la forma esperada, se prescinde de ellos en la plataforma, ya que influyen de forma notable en la latencia y el consumo de recursos lógicos del sistema, aumentando ambos parámetros.

Una vez resumidas las características de todos los bloques IP necesarios para que el sistema cumpla con la funcionalidad deseada, en la Figura 42 se muestra la estructura final de la plataforma implementada. En este caso, es necesario señalar que en la versión final de la plataforma se han eliminados los ILAs ya que, tras verificar el correcto funcionamiento del sistema completo, no tiene mucho sentido mantenerlos, debido a que inciden en un exceso del consumo de recursos lógicos, así como en una disminución del *slack* disponible y, por tanto, aumentando la latencia de nuestro sistema.

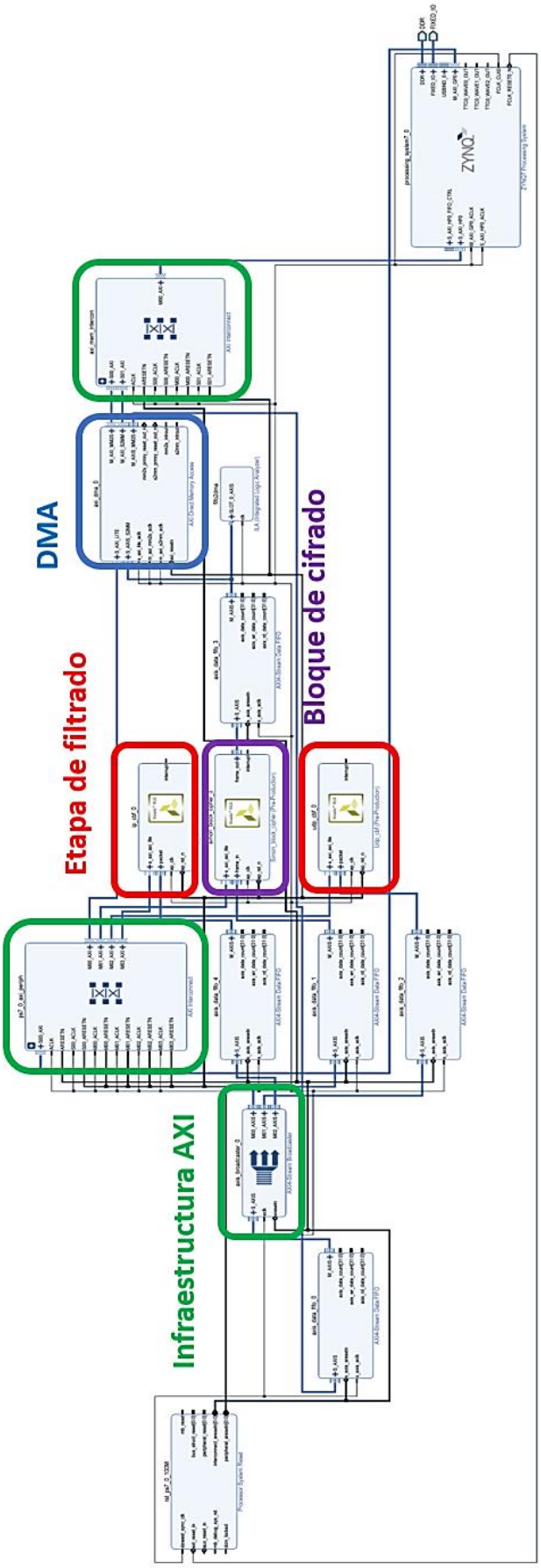



**Figura 42. Estructura de la plataforma final.**

## 5.3. Flujo de trabajo

El funcionamiento del sistema (Figura 43) se inicia desde el PS, que informa al DMA de que dispone de paquetes de datos a transmitir y que se encuentran almacenados de forma temporal en la memoria DDR externa al chip (para su acceso y gestión, es necesario definir la interfaz de E/S salida oportuna). Una vez que se le señaliza al DMA la existencia de datos disponibles, éste accede a la memoria, sin necesidad de intervención de la CPU, y transmite la información obtenida hacia el AXI4-Broadcaster.

En este punto, el Broadcaster envía la información recibida de forma simultánea a través de sus tres interfaces AXI4-Stream de salida, llegando el mismo paquete de datos tanto a los filtros como al bloque de descifrado Simon. Para el caso concreto del sistema a desarrollar, solo incluiremos la parte de descifrado, ya que el único sentido de comunicación contemplado es desde un *host* externo conectado a la nube hacia el nodo *Fog*. Si, en futuras ampliaciones del proyecto, se decide realizar el sentido de comunicación inverso (desde el nodo *Fog* hacia el exterior), se hace también obligatorio el uso del bloque de cifrado.

Debido a que el *software* empotrado, que se comentará con detalle más adelante, se ejecuta (aunque sea con mínimas diferencias) de forma secuencial, se ha establecido como primera ejecución la etapa de filtrado. De este modo, en caso de que el resultado del análisis sea negativo (es decir, que la dirección IPv6 de origen o el puerto UDP de destino no estén incluidos en las listas gestionadas por los filtros), se pasa a descifrar el *payload* del paquete Ethernet recibido. En caso de que el resultado del análisis realizado por alguno de los filtros sea positivo, se descarta automáticamente el paquete recibido y se evita aplicar la etapa de descifrado, reduciendo de esta forma el tiempo de ejecución del sistema completo.

En este punto, cabe pensar que, debido a que los bloques *custom* no pueden ejecutarse de forma concurrente, no presenta una ventaja significativa tener los datos disponibles a la entrada de ambos bloques IP al mismo tiempo. Sin embargo, de este modo se consigue evitar un acceso a memoria adicional para proporcionar la información al bloque IP con menor prioridad (en este caso, el bloque de descifrado), siendo este tipo de transacciones los cuellos de botella típicos de un sistema basado en memoria y que mayor latencia introducen en el sistema.

En caso de que el resultado del filtro sea positivo, se informa al PS de que el paquete Ethernet recibido no cumple las políticas de seguridad definidas, siendo función de este último proceder a descartar dicho paquete. Por otra parte, si el resultado es negativo, el bloque de descifrado actuará sobre el *payload* recibido y lo devolverá hacia el DMA, el cual realizará su transmisión hacia la memoria DDR, para que sea accesible por parte del PS para que realice con él las acciones que se consideren oportunas.

Es importante resaltar que el filtro en ningún momento debe esperar a que se reciba el *payload* del paquete de datos entrante, ya que su función se limita a analizar campos concretos de la cabecera. Asimismo, el bloque de descifrado no debe modificar, en ningún caso, la cabecera, ya que se modificarían sus rutas de encaminamiento y daría lugar a una comunicación errónea, ya que el paquete nunca llegaría a su destino.

En relación a las interfaces AMBA AXI4 empleadas, la comunicación entre el PS y el DMA precisa de una interfaz tipo Memory-Mapped, asignando de esta forma al DMA un rango de direcciones de memoria, tanto de escritura como de lectura, sobre las que actuar. Todas las interfaces vinculadas a puertos de configuración necesarios para gestionar los parámetros de los distintos bloques IP que conforman la plataforma, se han definido como AXI4-Lite. Finalmente, todo el flujo de información que recorre el sistema de módulo en módulo se propaga a través de interfaces AXI4-Stream orientadas a una arquitectura tipo *dataflow* como la que centra nuestra atención en este trabajo.

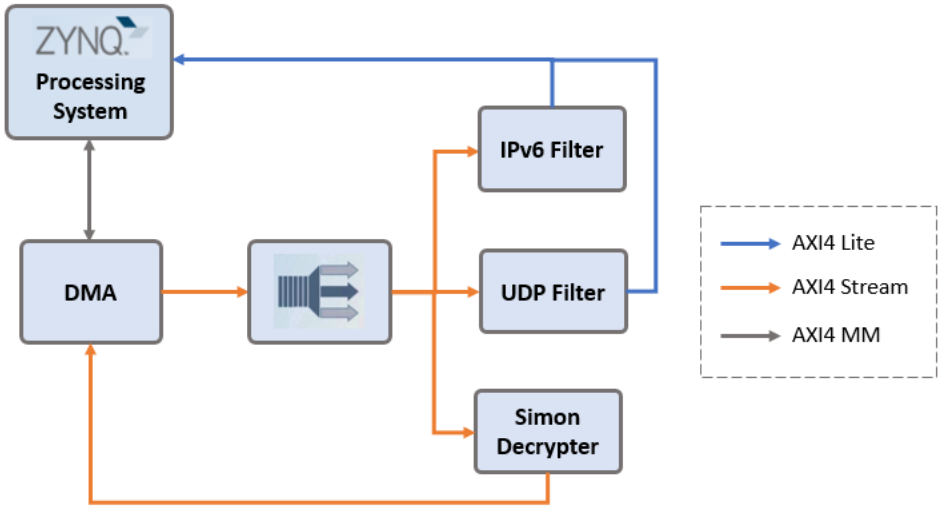



**Figura 43. Diagrama de bloques del sistema final.**

## 5.4. Síntesis lógica e implementación

Una vez definida la plataforma, es el momento de analizar los resultados obtenidos durante las etapas de síntesis lógica e implementación. En referencia a la latencia, se consigue obtener finalmente un WNS de 0,277 ns para una frecuencia de reloj de 150 MHz, situándose la ruta crítica del sistema en los accesos a memoria realizados por el bloque de descifrado, el cual limita la frecuencia máxima de trabajo de la plataforma, al ser el bloque que precisa de un mayor periodo de reloj para su correcta operación. El hecho de que el *slack* sea positivo nos indica que, en la medida que indica su valor, es posible reducir el tiempo de ciclo del sistema. Sin embargo, el PLL que incluye el SoC no es lo suficientemente preciso como para establecer frecuencias de trabajo concretas, por lo que no se puede incrementar la frecuencia de trabajo hasta el siguiente valor que es capaz de generar (187 MHz). Por su parte, la frecuencia de los procesadores ARM Cortex-A9 se ha fijado a 666,66 MHz, la máxima permitida.

Además, hay que destacar que, para alcanzar estos valores de frecuencia de trabajo, se ha aplicado una estrategia de optimización de *retiming* durante la etapa de síntesis lógica, que busca reducir la longitud de las rutas críticas del sistema. Asimismo, la inclusión de las FIFOs en el diseño también colabora a mejorar el *slack*, ya que evitan los citados cuellos de botella en las comunicaciones entre bloques IP que trabajan a diferentes velocidades de procesamiento. También es necesario indicar que, aunque los bloques IP modelados en alto nivel son capaces de alcanzar frecuencias de trabajo superiores, hay ciertas partes de la plataforma que limitan su frecuencia máxima, como son la interfaz AXI4-Lite o el bloque DMA, que no superan los 200 MHz de frecuencia de funcionamiento.

Analizada la frecuencia de trabajo máxima del sistema, se pasa a continuación a analizar el consumo de recursos derivado de su implementación sobre la lógica programable de la FPGA. Los resultados obtenidos se reflejan en la Tabla 13 y, de forma más gráfica, en la Figura 44, a partir de los cuales se puede concluir que el consumo de recursos del sistema es reducido, no superando para ninguno de los distintos recursos lógicos disponibles el 20% del total y disponiendo de la suficiente área libre para poder incluir mayor funcionalidad al sistema en caso de que se considere oportuno en posteriores ampliaciones del proyecto.

**Tabla 13. Consumo de recursos de la plataforma.**

| IP block | Slices | LUTs | FFs | BRAMs |
|---|---|---|---|---|
| **DMA** | 5,14% | 2,94% | 1,96% | 3,57% |
| **AXI infrastructure** | 5,98% | 3,62% | 2,07% | 0% |
| **FIFOs** | 1,74% | 0,70% | 0,65% | 5,35% |
| **UDP filter** | 1,27% | 0,69% | 0,57% | 0,36% |
| **IPv6 filter** | 3,11% | 1,50% | 1,31% | 1,79% |
| **Simon decypter** | 2,58% | 1,93% | 0,83% | 2,86% |
| **TOTAL** | **19,82%** | **11,38%** | **7,39%** | **13,93%** |

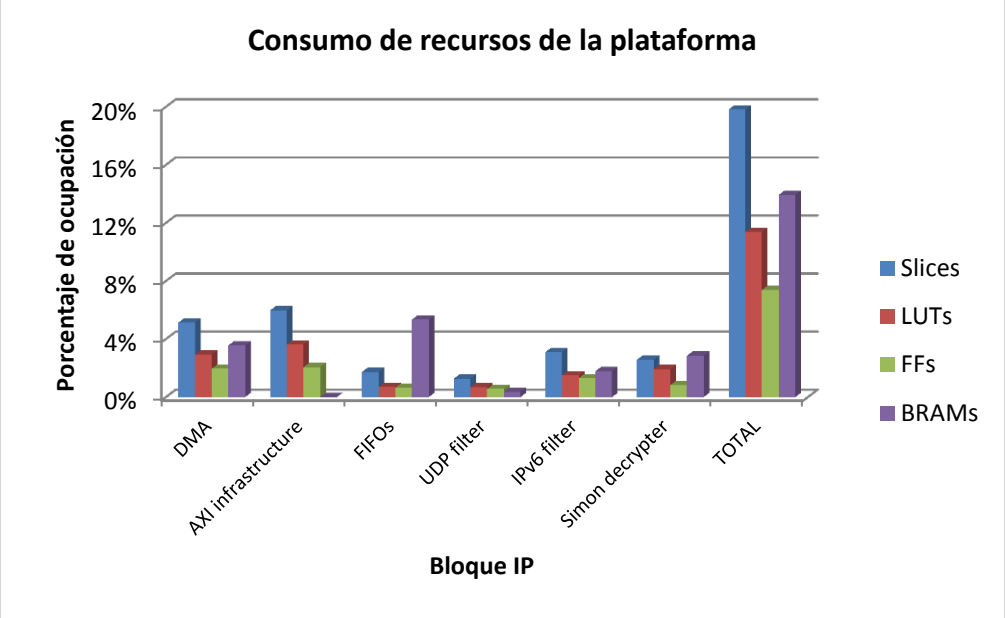



**Figura 44. Consumo de recursos de la plataforma.**

Con el fin de tener una imagen orientativa del consumo de recursos derivado del *place & route* de la plataforma definida sobre la lógica programable disponible en el PL, se incluye en la Figura 45 el *layout* del dispositivo Zynq-Z7020 con el sistema mapeado. Resaltados en color amarillo se observan los filtros para IoT modelados y en rosa el bloque de descifrado Simon.

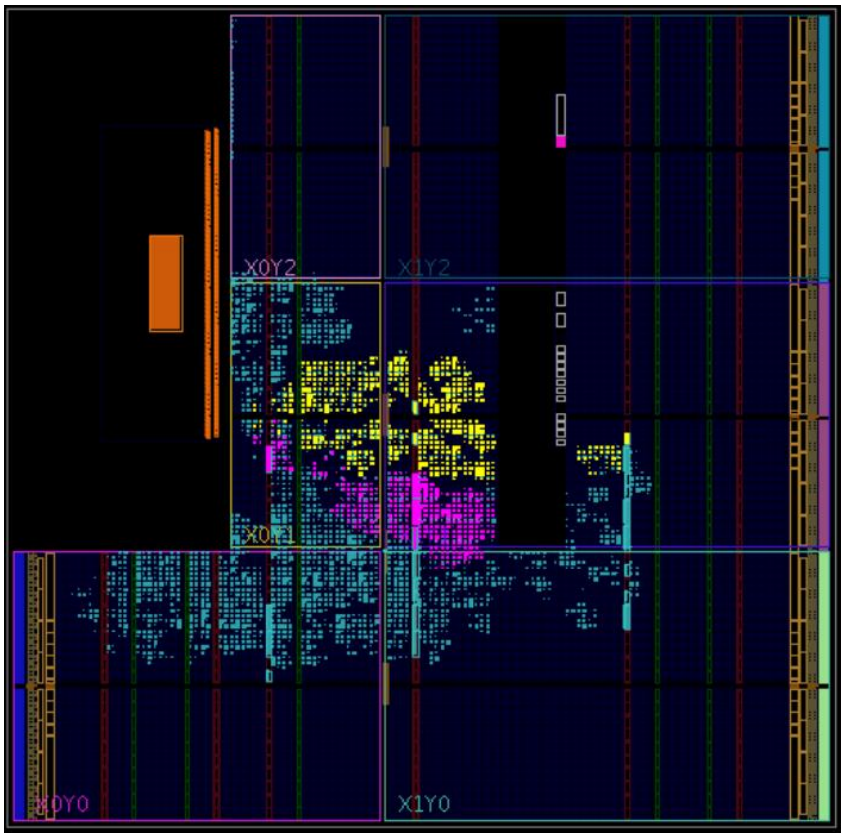



**Figura 45. Layout del dispositivo Zynq con el sistema mapeado.**

Finalmente, un último parámetro que resulta clave analizar en una aplicación orientada a *low power* como es el Internet de las Cosas es el consumo de energía del sistema. Para ello, se ha obtenido el consumo de potencia dinámica (hay que recordar que la potencia estática no depende del diseño realizado) sin aplicar ningún tipo de optimización y, posteriormente, se ha aplicado una estrategia de optimización orientada a reducir el consumo de energía (*power_opt_design*) durante la fase de implementación de la plataforma. Además, en ambos casos se ha descontado el consumo derivado del bloque de procesamiento ya que, al ser su inclusión necesaria en toda plataforma, no se considera un consumo derivado del diseño llevado a cabo

Hay que resaltar que la estrategia que sigue Vivado para reducir el consumo de potencia dinámica del sistema es la aplicación de *clock gating* a todos los bloques que componen la plataforma. Esta técnica se basa en la incorporación de lógica extra para poder interrumpir, a la entrada de cada módulo de la jerarquía, la señal de reloj procedente del *clock tree*. De este modo, se evita que los registros integrados en estas partes del diseño conmuten, limitándose de esta forma el consumo energético a las corrientes de fuga de los transistores que lo integran.

Siguiendo estas directrices, se han obtenido para ambos casos (con y sin estrategia de optimización) los resultados mostrados en la Figura 46, donde se puede apreciar que la reducción del consumo global de la plataforma es de un 1,1%, por lo que la aplicación de la estrategia de optimización disponible en Vivado no tiene un efecto considerable sobre el sistema.

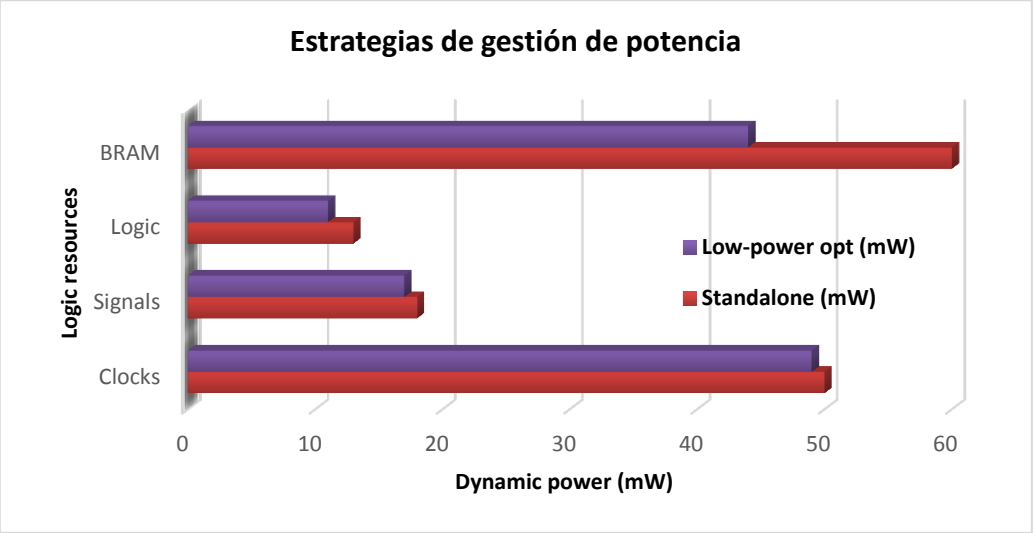



**Figura 46. Comparación del consumo de potencia del sistema según la estrategia aplicada.**

La aplicación de la estrategia de optimización orientada a *low-power* repercute tanto sobre la frecuencia de trabajo como sobre el consumo de recursos lógicos del sistema ya que, para reducir el consumo energético, la herramienta realiza una redistribución del mapeado del sistema durante la etapa de *place & route.* En referencia a la frecuencia de trabajo, aunque se sigue manteniendo a 150 MHz, al aplicar la estrategia de optimización señalada se dispone de un *slack* menor (0,15 ns). Por otro lado, tal como se observa en la Tabla 14 y de forma gráfica en la Figura 47, los cambios en la cantidad de recursos lógicos consumidos en función de la estrategia de optimización para consumo de energía aplicada no son significativos, por lo que no se trata de un parámetro que marque la decisión de optar por una estrategia u otra. En cuanto a la etapa de *place & route*, hay que señalar que, aunque se realiza una redistribución de los recursos sobre el dispositivo Zynq, el mapeado no sufre demasiados cambios, localizándose el diseño completo aproximadamente en el mismo sector del dispositivo.

**Tabla 14. Consumo de recursos lógicos en función de la estrategia de optimización aplicada.**

| | Standalone | Low-power opt |
|---|---|---|
| **Slices** | 19,82% | 20,05% |
| **LUTs** | 11,38% | 11,30% |
| **FFs** | 7,39% | 7,10% |
| **BRAMs** | 13,93% | 13,57% |

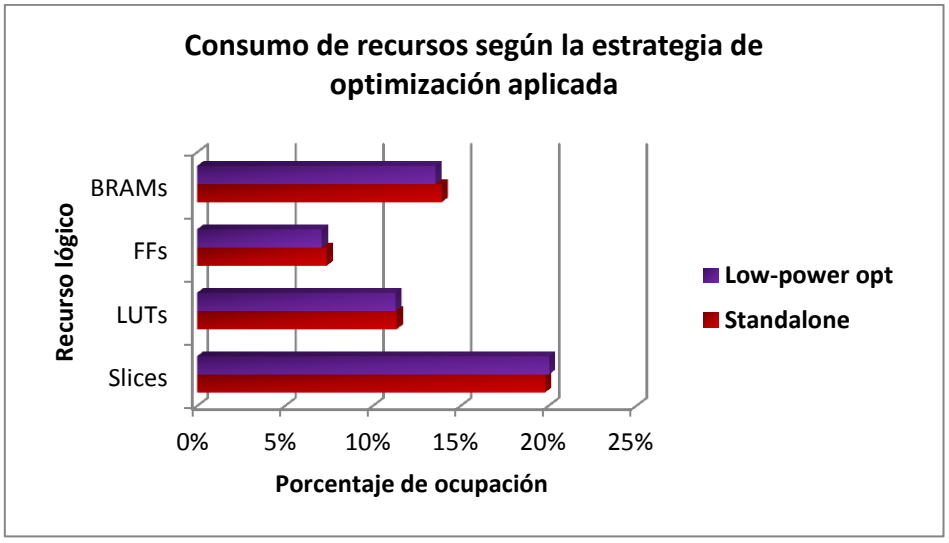



**Figura 47. Consumo de recursos según la estrategia de optimización aplicada.**

## 5.5. Diseño del software empotrado

Una vez montada la plataforma completa, integrando los bloques modelados en alto nivel con los propios de Xilinx para cumplir con la funcionalidad completa del sistema, el siguiente paso consiste en el diseño de la aplicación *software* que se encargará de gestionar su correcto funcionamiento.

### 5.5.1. Flujo de diseño

Partiendo de la descripción *hardware* de la plataforma creada y exportada desde Vivado, se posibilita la creación en Vivado SDK del BSP (*Board Support Package*), una colección de *drivers* que permiten al diseñador interactuar, mediante *software*, con aquellos módulos incluidos en la plataforma, así como establecer comunicaciones con elementos ajenos al dispositivo e incluidos en la placa de prototipado mediante la ejecución de los controladores oportunos. Asimismo, incluye funciones específicas (inicialización, selección del modo de ejecución, gestión de las listas de acceso, etc.) para los bloques *custom* diseñados y vinculadas a la interfaz AXI4-Lite definida como puerto de configuración en

todos ellos. De esta forma, se facilita enormemente la integración *hardware/software* del sistema completo, permitiendo una reducción de los tiempos de diseño respecto a los plazos que precisaría el diseñador para modelar toda esta funcionalidad.

Creado el BSP, es turno de realizar la aplicación *software* correspondiente al diseño de la plataforma, de forma que su integración resulte satisfactoria. En este caso, la aplicación se compondrá, además de la etapa de inicialización y configuración de los bloques *custom* y del DMA, de la función principal del sistema.

Esta función comenzará con la evaluación, por parte de la etapa de filtrado, de la cabecera del paquete Ethernet entrante al nodo *Fog*. En caso de devolver un resultado positivo de su análisis (esto es, que o bien la dirección IPv6 de origen se encuentra en la *blacklist* generada o que el puerto UDP al que está destinada la solicitud de conexión no está registrada en la *whitelist* configurada), se omitirá el descifrado del *payload* del paquete recibido. En caso contrario (ambos filtros devuelven una respuesta negativa tras la inspección realizada), se realiza la llamada al bloque de descifrado Simon, que pasa a descifrar el *payload* del paquete y devuelve el mensaje original al DMA, de forma previa a ser almacenado en la memoria DDR *off-chip* y a la cual tendrá acceso el PS para realizar las acciones necesarias sobre dicho mensaje.

El proceso de compilación permite obtener el ejecutable en formato ELF necesario para programar el PL del dispositivo Zynq, que se descarga en la placa de prototipado ZedBoard a través de su puerto de configuración JTAG.

En caso de que se dé la situación de que el sistema no responda de la forma esperada, se hace necesaria una etapa de depurado del código, apoyándonos tanto en los mensajes de estado y control de errores incluidos en la aplicación, como en el analizador lógico proporcionado por los ILAs conectados a aquellas líneas de la plataforma que se consideran críticas para realizar el seguimiento del correcto funcionamiento global del sistema. Una vez localizado el error, se debe proceder a enmendarlo, proceso que puede tener que realizarse en cualquiera de las etapas de que consta el flujo de diseño global del sistema (modelado en alto nivel, diseño de la plataforma o desarrollo de la aplicación *software*).

En la Figura 48 se muestra el diagrama de bloques del flujo de diseño descrito anteriormente para completar la integración *hardware/software* del sistema:

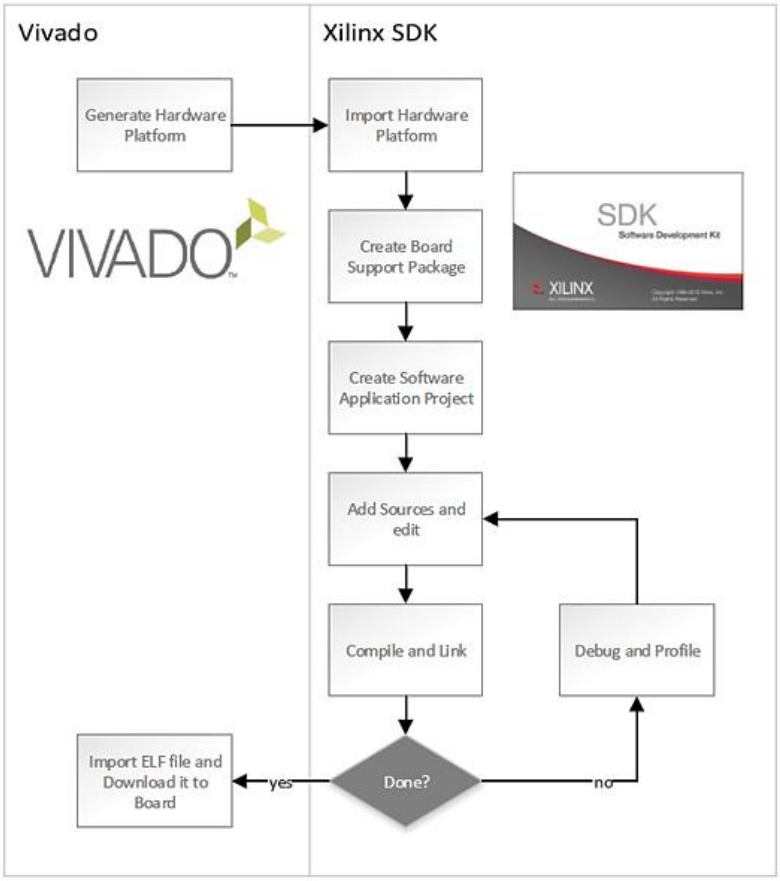



**Figura 48. Flujo de diseño para la integración hardware/software del sistema.**

## 5.5.2. Gestión del DMA

Tal como se ha hecho referencia en el apartado de la Arquitectura de la plataforma, el DMA resulta ser un bloque clave para que el sistema completo de comporte de forma correcta, ya que es el módulo que se encarga de gestionar los paquetes Ethernet recibidos a través de la red y almacenados en memoria de forma previa a su tratamiento, para posteriormente alimentar a los bloques *custom* incluidos en la plataforma, con el objeto de que realicen la funcionalidad para la que fueron concebidos. Por tanto, si el comportamiento del DMA no es el correcto, se producirán problemas en la plataforma derivados de la falta de comunicación entre el PS y los bloques IP de filtrado y descifrado.

Como todo bloque IP que vaya a ser gestionado desde la aplicación *software*, el DMA necesita ser inicializado y configurado de forma previa a toda comunicación a realizar. Para ello, se hace uso de una serie de funciones predefinidas que están incluidas en el fichero

*xdma.h* del BSP generado. En este punto, el diseñador debe establecer una serie de parámetros, como son el tamaño de los *buffers* de transmisión y recepción del módulo o el tamaño máximo de las transmisiones a realizar. Para esta aplicación, se han definido los *buffers* como *arrays* de 400 palabras de 32 bits ya que, como se mencionó durante el proceso de diseño de los bloques IP, esta dimensión se corresponde al tamaño máximo que puede presentar un paquete Ethernet.

Además, en esta fase se desactivan las interrupciones en el DMA, ya que el modo de operación elegido ha sido *polling*, a la espera de que desde el PS se informe de que existen datos disponibles que transmitir hacia el resto del sistema. Esta decisión se fundamenta en el hecho de que nuestro sistema estará ocioso mientras no haya existencia de datos que transmitir, por lo que no tiene sentido suspender el proceso de ejecución ante la llegada de nuevos datos.

La función de transferencia, por su parte, comienza con la copia del paquete recibido en el *buffer* de salida del DMA. De forma previa a la realización de la transferencia, es necesario realizar un *flush* de la memoria caché, con el fin de eliminar datos de transferencias anteriores que puedan corromper la integridad del nuevo mensaje a enviar.

Realizados estos pasos, se realiza la transmisión del paquete desde el DMA tanto a la etapa de filtrado como al bloque de descifrado (a través del AXI4-Stream Broadcaster). A esta función únicamente es necesario pasarle la instancia creada para el DMA, así como el *buffer* donde están almacenados los datos, el tamaño de la transferencia y el sentido de comunicación (en este caso, desde el DMA hacia los bloques *custom*).

```
int dma_transfer() {

	int status = 0;

	for (int i = 0; i < WORD_TOTAL; i++)
		dma_in_buf[i] = 0, dma_out_buf[i] = 0;

	// Copy the frame into the DMA output buffer
	for (int i = 0; i < WORD_TOTAL; i++)
		dma_out_buf[i] = frame_in[i];

	// Flush the SrcBuffer before the DMA transfer, in case the Data Cache is
enabled
	Xil_DCacheFlushRange((INTPTR) dma_in_buf, FRAME_SIZE);
	Xil_DCacheFlushRange((INTPTR) dma_out_buf, transfer_length);

	// Kick off MM2S transfer
```

```
	status = XAxiDma_SimpleTransfer(&axi_dma, (UINTPTR) dma_out_buf,
			transfer_length,XAXIDMA_DMA_TO_DEVICE);

	if (status != XST_SUCCESS) {
		xil_printf("ERROR! Failed to kick off MM2S transfer!\n\r");
		return XST_FAILURE;
	}

	// Wait for transfers to complete
	while (XAxiDma_Busy(&axi_dma, XAXIDMA_DMA_TO_DEVICE)) {
		/* Wait */
	}

	xil_printf("DMA transfer complete!\n\r");

	return 0;
}
```

### 5.5.3. Modelado de la función principal

La función principal que gobierna el funcionamiento del sistema completo se inicia con una etapa de configuración de los filtros, componiendo de esta forma la *blacklist* que controlará el filtro para direcciones IPv6 y la *whitelist* que gestiona el filtro para puertos UDP. Además de gestionar las listas, se hace necesaria la inicialización de ambos filtros, conformándose los *arrays* que integran las estructuras CBF y las variables de control de los bloques IP.

Posteriormente, se realiza la búsqueda en los campos correspondientes de la cabecera del paquete Ethernet entrante por parte de ambos filtros. En caso de que al menos uno de ellos devuelva un resultado positivo (esto es, un 0), se concluye la ejecución del sistema, al haberse detectado una posible amenaza que comprometería la integridad del nodo *Fog*. Por otro lado, si ambas respuestas son negativas (valor 1 como respuesta), se solicita la devolución del *payload* descifrado por parte del DMA, el cual es posteriormente almacenado en memoria a la espera de que el PS se encuentre disponible para realizar sobre él la gestión correspondiente.

```
int system_app() {

	int status = 0, status_ip = 0, status_udp = 0;

	// IP fields to insert
	u32 network_1_1 = 1073612039;
	u32 network_1_2 = 1;
	u32 network_2_1 = 265905879;
	u32 network_2_2 = 605126000;
	u32 node_1_1 = 33588991;
```

```
	u32 node_1_2 = 4261773530;

	// UDP ports to insert
	u32 port_1 = 3345;
	u32 port_2 = 22;
	u32 port_3 = 80;

	status = filter_ip_performance(INIT, NULL, NULL, UNKNOWN, NULL);

	if (status != 0) {
		xil_printf("Test failed. IPv6 couldn't be created\r\n");
		return XST_FAILURE;
	}
	status = filter_udp_performance(INIT, NULL, NULL);

	if (status != 0) {
		xil_printf("Test failed. UDP filter couldn't be created\r\n");
		return XST_FAILURE;
	}
	// Filter's operations
	status = filter_ip_performance(ADD, network_1_1, network_1_2,
NETWORK_ID,NULL);
	status = filter_ip_performance(ADD, node_1_1,node_1_2, NODE_ID, NULL);
	status = filter_ip_performance(ADD, network_2_1,network_2_2, NETWORK_ID,
NULL);
	status = filter_ip_performance(ADD, node_2, NODE_ID, NULL);
	status = filter_udp_performance(ADD, port_1, NULL);
	status = filter_udp_performance(ADD, port_2, NULL);
	status = filter_udp_performance(ADD, port_3, NULL);

	status_ip = filter_ip_performance(SEARCH, NULL, NULL, UNKNOWN, frame_in);
	status_udp = filter_udp_performance(SEARCH, NULL,frame_in);

	status = status_ip & status_udp;

	if (status != 1) {
		xil_printf("A potential threat has been detected\r\n");
		return XST_FAILURE;
	}

	//Simon decrypter response
	status = XAxiDma_SimpleTransfer(&axi_dma,(UINTPTR) dma_in_buf, FRAME_SIZE,
	XAXIDMA_DEVICE_TO_DMA);

	// Wait for transfers to complete
	while (XAxiDma_Busy(&axi_dma, XAXIDMA_DEVICE_TO_DMA)) {
		/* Wait */
	}

	if (status != 0) {
		xil_printf("Test failed. The packet couldn't be decrypted!! \r\n");
		return XST_FAILURE;
	}

	return XST_SUCCESS;
}
```

## 5.6. Conclusión

En este capítulo se ha descrito el proceso de integración de los bloques *custom* modelados en alto nivel con el resto de módulos que componen la plataforma final, resaltando las características elementales de cada uno de ellos y sus interfaces para establecer comunicaciones con el resto del sistema.

Asimismo, se ha realizado un breve estudio sobre las prestaciones alcanzadas, tanto en términos de frecuencia de trabajo como en consumo de recursos lógicos y de potencia, en función de la estrategia de optimización aplicada. Finalizado dicho análisis, se concluye que la utilización de la estrategia de optimización vinculada a *low-power* no presenta un cambio significativo en ninguno de los parámetros analizados respecto a la plataforma *standalone* (es decir, sin ningún tipo de optimización aplicada).

Finalmente, se ha explicado el *software* empotrado que ha sido necesario desarrollar para gobernar el correcto funcionamiento del sistema, resaltando especialmente el flujo de diseño que es necesario seguir para cumplir con esta tarea de forma satisfactoria.

# Capítulo 6. Validación del sistema y resultados

## 6.1. Introducción

En este capítulo se presenta la validación de la plataforma, de sus principales bloques IP en cuanto a su funcionalidad y a su respuesta, incluyendo tanto los resultados de latencia y *throughput* individuales de los bloques *custom* modelados como los obtenidos para el sistema completo.

## 6.2. Prestaciones a medir

Los cálculos temporales a determinar, la latencia y el *throughput*, se realizan sobre el prototipo del sistema ya sea a nivel de señales a las que se accede de forma virtual en el subsistema *hardware*, apoyándonos en la herramienta Hardware Manager de Vivado (proporciona la visualización de los diagramas de señales en aquellas líneas de la plataforma donde se ha conectado un ILA), como en *software*, haciendo uso de la librería *xtime_l.h* de Xilinx, que permite introducir marcas temporales en la aplicación empotrada con el fin de medir sus prestaciones. Para que el Hardware Manager actúe, es necesario conectar el *host* a la placa de prototipado (en nuestro caso, la ZedBoard) a través del puerto JTAG disponible mediante una conexión USB 2.0.

Los parámetros concretos a medir sobre el diseño serán:

1. Latencia en *hardware* de la etapa de filtrado.
2. Latencia en *hardware* del bloque de cifrado.

3. Latencia, tanto en *hardware* como en *software*, de la plataforma completa. En este caso, hay que contemplar dos situaciones posibles: aquella en la que los filtros devuelven un resultado positivo y descarta el paquete, y un segundo caso en el que retorna un negativo y el *payload* del correspondiente paquete de datos es descifrado por el bloque Simon.

Determinados los valores de la latencia en los distintos instantes del flujo de datos, se procede a calcular las prestaciones, tanto de los bloques *custom* como del sistema completo, en términos de *throughput*. Es decir, qué capacidad de transmisión de información tienen por unidad temporal. De este modo, podemos concluir qué estándares IEEE 802.3 de los planteados en el apartado de Objetivos se han conseguido cubrir con el diseño realizado.

En cuanto al consumo de recursos lógicos mostrado en el apartado de Síntesis lógica e implementación de la plataforma, la información obtenida en este sentido es bastante útil para determinar el factor de la utilización de las *slices* de la FPGA, que indica la calidad de la etapa de implementación realizada por el entorno Vivado. Este parámetro se determina teniendo en cuenta el número de LUTs y de FFs de las que se dispone por *slice* en el dispositivo Xilinx Zynq, que es de 4 y 8, respectivamente.

## 6.3. Metodología de validación

Para validar el sistema completo se hará uso de una base de datos con la que se cuenta en la división de proyectos anteriores, y que incluye un conjunto de paquetes Ethernet previamente extraídos de la red haciendo uso del *sniffer* Wireshark. De este modo, se cuenta con datos cuyos parámetros (cabecera y *payload*) son conocidos, pudiendo predecir el comportamiento que deben presentar las distintas partes que conforman el sistema, así como su funcionamiento global. En este punto, hay que resaltar que la obtención de los paquetes para IPv6 no es trivial, ya que no abundan aquellos que emplean IPv6 como protocolo de red y UDP como protocolo de transporte fuera de un entorno IoT como es una red LAN común.

Por tanto, los parámetros que resulta clave conocer son la dirección IPv6 de origen, el puerto UDP de destino, el campo de *Next Header* para determinar qué protocolo de transporte se está utilizando en la conexión establecida y el campo de longitud del *payload*,

para conocer la cantidad de palabras de 32 bits de datos que debe descifrar el bloque basado en el algoritmo Simon.

## 6.4. Resultados obtenidos

En primer lugar, se procede a medir la latencia de los filtros. Se considera la latencia total de ejecución al tiempo que transcurre desde que se recibe la primera palabra de 32 bits de la cabecera del paquete hasta que el bloque IP devuelve una respuesta sobre la consulta realizada.

Para su cálculo nos apoyaremos, tal como se ha mencionado, en el diagrama de formas de onda proporcionado por el Hardware Manager de Vivado. De forma previa a lanzar el sistema, ha sido necesario marcar mediante la opción *Mark Debug* aquellas señales internas de los bloques que se consideran de interés para comprobar su correcto funcionamiento y, posteriormente, crear un núcleo de depurado que permitirá su visualización apoyándose en los ILAs disponibles en la infraestructura IP de Xilinx.

Para realizar la medida, se situarán dos marcadores en los puntos de inicio y final de la operación de los filtros, obteniendo de esta forma la cantidad de ciclos de reloj que precisa para finalizar su funcionalidad. Sabiendo que la frecuencia máxima de trabajo alcanzada ha sido de 150 MHz, multiplicando el número de ciclos por el tiempo de ciclo (6,67 ns) se obtiene la latencia de los filtros. Para el caso del bloque de descifrado, el modo de proceder será exactamente el mismo.

Aplicando este procedimiento, se obtiene que el filtro UDP precisa de 80 ciclos para finalizar su operación de consulta, lo que supone, asumiendo un tiempo de ciclo de 6,67 ns, una latencia final de **0,534 µs**. Además, en la Figura 49 se puede observar el valor de otras señales relevantes del bloque IP, como son el valor de la variable *detection*, que determina si el valor del campo *Next Header* de la cabecera es el correspondiente a UDP, o el puerto UDP que ha sido extraído del paquete en cuestión.

De forma equivalente, para el filtro IPv6 se necesitan 120 ciclos para concluir un método de consulta (Figura 50), ya sea acerca de una subred o de un nodo concreto. Teniendo en cuenta este valor y el tiempo de ciclo del sistema, se obtiene una latencia para este caso de **0,800 µs**.

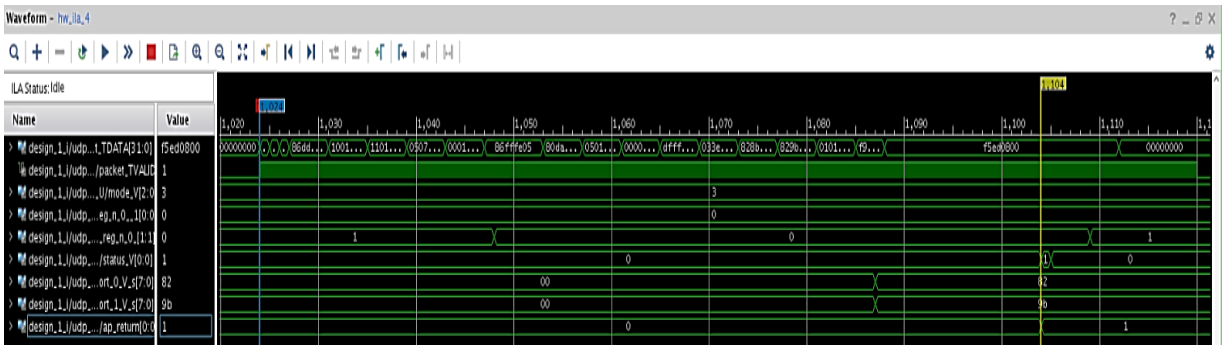


**Figura 49. Cálculo de la latencia del filtro UDP.**

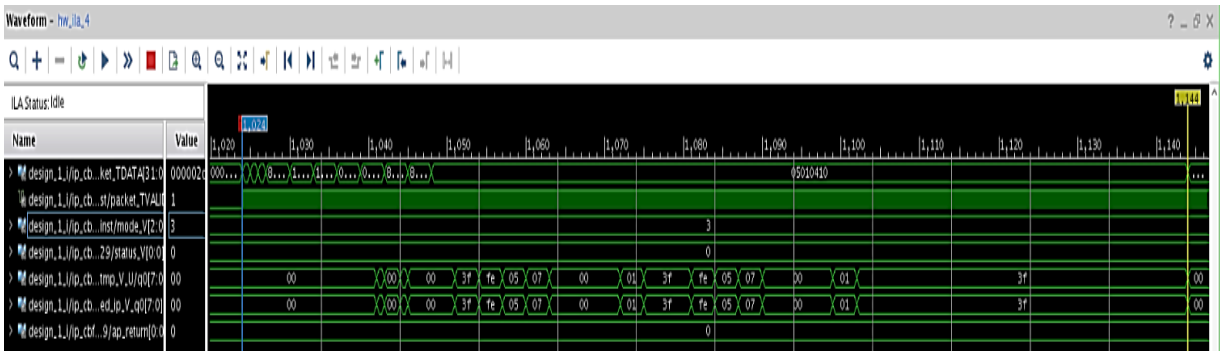


**Figura 50. Cálculo de la latencia del filtro IPv6.**

El bloque de descifrado (Figura 51), tal como se comentó en el apartado correspondiente a su diseño, es el que delimita la frecuencia máxima del sistema, al tener la latencia más alta de los bloques IP que integran el diseño. En concreto, toma 1.083 ciclos en acabar el proceso de descifrado del *payload*, desde que comienza a recibir el paquete hasta que pone en la interfaz AXI4-Stream de salida la última palabra de 32 bits tratada. Esto da como resultado una latencia de **7,224 µs**, hasta una orden de magnitud mayor que la de los filtros.

**Figura 51. Cálculo de la latencia del bloque de descifrado.**

A partir de los valores de latencia obtenidos para los bloques IP de forma independiente, se puede deducir la latencia total de la plataforma en *hardware*. De esta

forma, su latencia será de **1,334 µs**, correspondiente a la suma de las latencias de los filtros, en el caso de que el análisis de al menos uno de ellos sea positivo. En caso de que ambos devuelvan una respuesta negativa, la latencia total del sistema será la suma de la de los filtros más la del bloque de descifrado. Esto es, una latencia total de **8,558 µs**.

Para la medida de la latencia en *software* es necesario, tal como se ha comentado anteriormente, con incluir el fichero *xtime_l.h* en la aplicación y aplicar marcas temporales al comienzo y final de la función principal mediante la función *XTime_GetTime*. El resultado que devuelve la diferencia de dichas marcas temporales es la mitad del número de ciclos de CPU consumidos para finalizar la ejecución de la funcionalidad del bloque, por lo que es necesario multiplicar por 2 este valor para obtener el valor final de la latencia.

Nuevamente, obtendremos dos valores para la latencia, en función del caso que se presente en el sistema en función del resultado del análisis realizado por los filtros. Sabiendo que la frecuencia de trabajo máxima del microprocesador ARM Cortex-A9 integrado en el dispositivo Zynq para esta aplicación es de 666,66 MHz, obtenemos unos valores de latencia en *software* para el caso de que alguno de los filtros devuelva un positivo de **2,887 µs**, mientras que, para la funcionalidad completa del sistema, incluyendo la etapa de descifrado, obtenemos un valor de **2.067,460 µs**. Este valor pone nuevamente de manifiesto que la aceleración *hardware* de la etapa de cifrado resulta clave, ya que es aquella parte del sistema que presenta mayor carga computacional.

En la Figura 52 se recogen los valores de latencia obtenidos para los bloques IP modelados de forma individual y para los dos casos de operación posibles del sistema completo, tanto en *hardware* como en *software*. Debido a que el bloque de descifrado presenta una latencia de más de tres órdenes de magnitud que la etapa de filtrado, se ha escalado su valor con el fin de que la gráfica sea visualizada correctamente.

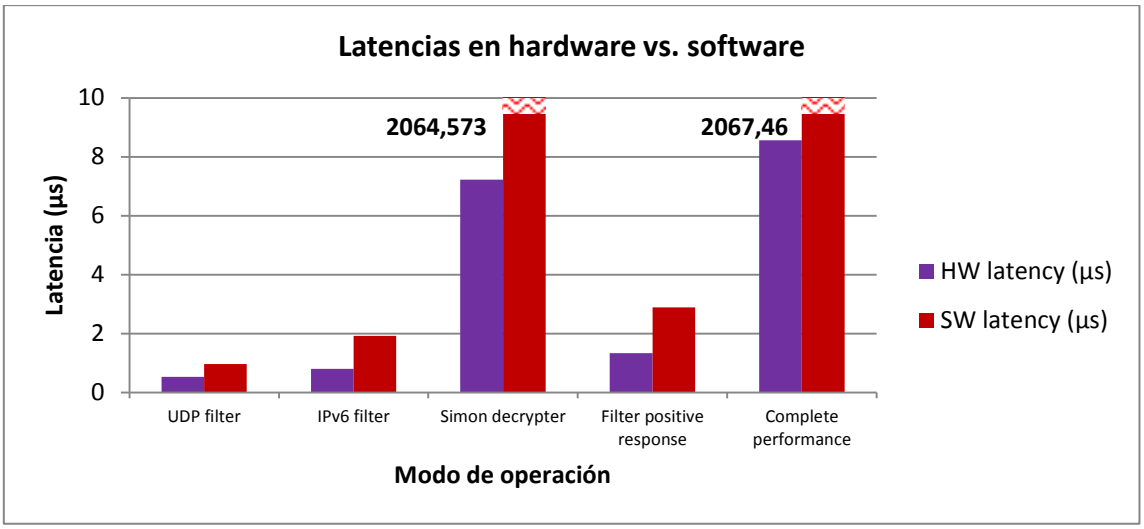



**Figura 52. Latencias en hardware vs software.**

Contando con los valores de la latencia tanto en *hardware* como en *software*, llegamos a la conclusión de que la solución *hardware* presenta un *speedup* de **x241** respecto a su homóloga en *software*.

El cálculo del *throughput* del sistema completo conocida la latencia del mismo resulta bastante intuitivo. Para ello, basta con dividir el tamaño de los paquetes empleados para realizar la validación (comprendidos entre 100 y 300 *bytes*) entre la latencia obtenida. Debido a que se han usado paquetes Ethernet de diverso tamaño, se realizará el cálculo con el valor promedio, obteniendo un *throughput* de:

$$Throughput = \frac{Packet_{size}}{Latency} = \frac{200 \cdot 8}{8{,}558 \cdot 10^{-6}} = \mathbf{187\ Mbps} \qquad (4)$$

Del resultado obtenido, se desprende que este sistema no cumplirá con ninguna de las especificaciones IEEE 802.3 planteadas en el apartado de Objetivos, lo cual se debe, tal como ya se ha hecho mención anteriormente en este documento, a las limitaciones que supone la incorporación del bloque de descifrado en el sistema. Sin embargo, tal como se planteó desde el inicio, la misión principal de este trabajo es asegurar las comunicaciones entrantes al nodo *Fog* consiguiéndose, al emplear el algoritmo Simon con 42 rondas, hacer el sistema robusto frente a la mayor parte de ataques que tienen lugar en una arquitectura IoT como la que tiene como objetivo este TFM, tal como se asegura en [61].

Para alcanzar *throughputs* mayores, habría que realizar un estudio exhaustivo de las optimizaciones que son posibles realizar sobre la implementación del algoritmo Simon

realizada, así como plantearse la posibilidad de llevar a *hardware* algún otro bloque de cifrado orientado a una implementación de este tipo de entre los mencionados en el apartado de Alternativas criptográficas.

Finalmente, señalar que en el supuesto de que alguno de los filtros devolviera una respuesta positiva acerca del análisis llevado a cabo sobre la cabecera del paquete Ethernet, no se realizaría la etapa de descifrado del *payload* y el sistema alcanzaría un *throughput* de **1,2 Gbps**, cumpliendo con los estándares IEEE 802.3z y IEEE 802.3ab para velocidades de 1 Gigabit Ethernet sobre fibra óptica y par trenzado.

Para concluir, se pasa a calcular el factor de utilización de las *slices* necesarias para implementar el diseño completo realizado. Partiendo de los datos referentes al consumo de recursos lógicos proporcionados por Vivado, obtenemos los siguientes FUs:

$$\mathrm{FU_{LUT}} = \frac{N_{LUTs}}{N_{slices}} = \frac{5838}{2525} = \mathbf{2,31} \quad (5) \qquad \mathrm{FU_{FF}} = \frac{N_{FFs}}{N_{slices}} = \frac{7415}{2525} = \mathbf{2,94} \quad (6)$$

Tal como se desprende de los resultados obtenidos, el factor de utilización de las LUTs se encuentra algo por encima de la mitad de su valor ideal, el cual es 4, debido a que gran parte de la lógica que compone el diseño requiere un tratamiento combinacional.

Por su parte, el factor de utilización de los *flip-flops*, el cual se deriva de la lógica secuencial incluida en el diseño, presenta un valor aproximado al 40% del ideal (8 en este caso), siendo un reflejo de la convivencia de distintas señales de control dentro de la plataforma y estando limitada cada *slice* al empleo de una única señal de control para todos los FFs que incluye en su rango de actuación. Hay que resaltar que en ningún caso se ha aplicado una estrategia de optimización agresiva en cuanto a eficiencia en la implementación se refiere, lo cual podría mejorar en cierta proporción los resultados obtenidos.

En la Tabla 15 se realiza una comparativa entre los resultados calculados en este trabajo para el FU con los obtenidos en otros trabajos de la división sobre dispositivos Zynq [63].

**Tabla 15. Comparativa del factor de utilización con los obtenidos en trabajos anteriores de la división.**

| | Tipo de diseño | Herramienta | Dispositivo | Factor de utilización | |
|---|---|---|---|---|---|
| | | | | LUTs | FFs |
| **Diseño 1** | Plataforma + IP | Vivado HLS | Zynq Z7020 | 2,21 | 2,90 |
| **Diseño 2** | IP | CtoS | Zynq Z7045 | 3,32 | 1,64 |
| **Diseño 3** | Plataforma + IP | Vivado HLS | Zynq Z7045 | 2,29 | 3,05 |
| **Diseño 4** | IP | Vivado HLS | Zynq Z7045 | 2,12 | 2,79 |
| **Este trabajo** | Plataforma + IP | Vivado HLS | Zynq Z7020 | 2,31 | 2,94 |

De la información obtenida, se concluye que para trabajos con características similares y empleando la misma herramienta de síntesis de alto nivel se obtienen resultados bastante similares, aunque ya se ha mencionado que este parámetro depende de la naturaleza del diseño. Asimismo, se aprecia que la herramienta CtoS de Cadence mejora sustancialmente el factor de utilización de las LUTs respecto a Vivado HLS, optimizando la gestión de la lógica combinacional del sistema.

## 6.5. Comparativas con otros trabajos

Debido a que en la literatura existente no se ha encontrado ningún trabajo en el que convivan en un único sistema etapas de filtrado y cifrado del tráfico Ethernet recibido para una aplicación orientada a Internet de las Cosas, se ha limitado a realizar una comparación de forma individual de las implementaciones de los filtros y del algoritmo Simon con trabajos ya realizados y disponibles en el estado del arte.

Para proceder a comparar el trabajo realizado con otras contribuciones similares en el mismo ámbito, hay que tener en cuenta el factor tecnológico, en cuanto a su implementación en diferentes tecnologías, debido a la generación a la que pertenecen.

En referencia a la etapa de filtrado, existen múltiples implementaciones en la literatura. Sin embargo, resulta especialmente complicado encontrar un denominador común sobre el cual realizar una comparación completamente objetiva.

Analizando el estado del arte, se han encontrado trabajos que, aunque se basan en modelos de memoria distintos y actúan en niveles diferentes del modelo OSI, presentan características similares al trabajo realizado en la etapa de filtrado de este TFM, al emplear durante la etapa de implementación dispositivos lógicos de Xilinx (Virtex-4 y 5) con prestaciones equiparables a las del dispositivo Zynq empleado. De este modo, en la Tabla 16 se resumen las principales características de los trabajos realizados en [77]–[79]. En este punto, hay que destacar que, al no proporcionarse en la mayoría de ellos el consumo de *slices* que ha supuesto la implementación del diseño sobre el dispositivo físico, no ha sido posible definir una ratio que permita realizar una comparativa equiparable entre ellos y el proyecto realizado y descrito en el presente documento.

La comparativa más precisa se podría realizar respecto a [79] debido a las similitudes existentes entre los trabajos tanto en la estructura de memoria empleada como en los dispositivos finales empleados para realizar la implementación del diseño. Sin embargo, su funcionalidad se limita únicamente a extraer el nombre del dominio HTTP a nivel de aplicación, lo que da lugar al *throughput* tan alto que alcanza. En este sentido, se considera que la etapa de filtrado diseñada en el presente TFM realiza un análisis más exhaustivo de los paquetes Ethernet entrantes, por lo que el *endpoint* localizado detrás está dotado de una mayor protección.

**Tabla 16. Comparativa de la etapa de filtrado con trabajos existentes en el estado del arte.**

| Trabajo | Dispositivo | Estructura empleada | Niveles OSI | Frecuencia (MHz) | *Throughput (Gbps)* |
|---|---|---|---|---|---|
| **Loinig *et al.*** [77] | Virtex-4 | Basado en RAM | 2 y 3 | 125,0 | 1,00 |
| **Rohrbeck, *et al.*** [78] | Virtex-5 | Basado en RAM | 2, 3 y 4 | 142,9 | 4,57 |
| **Yu, Cong, *et al.*** [79] | Virtex-5 | Bloom Filters | 7 | 272,0 | 8,70 |
| **Este trabajo** | Zynq Z7020 | Counting Bloom Filters | 3 y 4 | 229,0 | 1,20 |

En lo que respecta al bloque de descifrado Simon, en [59], [80], [81] se han realizado sendas implementaciones del algoritmo sobre FPGAs Spartan-6 de Xilinx que, aunque no pertenecen a la misma generación que la FPGA Artix-7 que incluye el dispositivo Xilinx Zynq

empleado, suponen un buen punto de comparación, debido a que sus frecuencias máximas de trabajo son similares (500 MHz para la Spartan-6 y 450 MHz para la Artix-7). En cuanto a los recursos lógicos, hay que tener en cuenta que la FPGA Artix-7 posee más del doble de *slices* que las Spartan-6 empleadas en los trabajos citados.

Asimismo, se incluye una comparativa con una implementación del algoritmo AES-128 [82], equiparable en los tamaños de palabra que maneja a la versión del algoritmo Simon realizada en el presente trabajo. En este caso, los dispositivos físicos seleccionados han sido los de la familia Virtex (versiones 5, 6 y 7), de mayores prestaciones que la Artix-7 incluida en el dispositivo Zynq Z-7020 empleado.

Realizadas estas aclaraciones, en la Tabla 17 se resumen las prestaciones principales tanto de los trabajos mencionados como del realizado en el presente TFM, haciendo hincapié en el lenguaje de modelado, el *throughput* alcanzado y el consumo de *slices* derivado de la implementación sobre la FPGA del diseño realizado, estableciendo una ratio entre estos dos últimos parámetros que sirva como medida de calidad entre los citados trabajos y el realizado en este proyecto, independientemente del proceso de fabricación CMOS al que pertenezca el dispositivo físico empleado.

**Tabla 17. Comparativa del bloque de descifrado Simon con trabajos existentes en el estado del arte.**

| Trabajo | Lenguaje | Throughput (Mbps) | Consumo *slices* | *Throughput/slices* |
|---|---|---|---|---|
| **Gulcan et *al.*** [59] | VHDL | 2,75 | 32/600 | 0,086 |
| **Gulcan et *al.*** [80] | Verilog | 3,60 | 36/600 | 0,100 |
| **Wetzels et *al.*** [81] | VHDL | 479,30 | 2.952/6.822 | 0,163 |
| **Jararweh et *al.*** [82] | VHDL | V5: 254,00<br>V6: 320,00<br>V7: 495,00 | V5: 3.060/3.120<br>V6: 2.963/11.640<br>V7: 4.380/51.000 | V5: 0,083<br>V6: 0,108<br>V7: 0,113 |
| **Este trabajo** | C/C++ | 221,00 | 372/13.300 | 0,594 |
| **Leyenda:** | V5: Virtex-5. V6: Virtex 6. V7: Virtex 7. | | | |

Tal como se desprende de los resultados obtenidos, la implementación llevada a cabo en este trabajo presenta la mejor ratio *throughput/slices* de la comparación realizada, tanto con las implementaciones de Simon presentes en el estado del arte como con la implementación del AES-128 realizada sobre la familia Virtex. En principio, cabría esperar

que una implementación realizada en lenguaje HDL resultara más eficiente que otra hecha en alto nivel, debido a que se tiene un mayor control sobre la lógica implementada, gestionando las señales a nivel BCA. Sin embargo, esta comparativa demuestra la calidad de las prestaciones derivadas de un modelado en C/C++ *hardware-friendly*, así como la eficiencia de las herramientas de síntesis de alto nivel y de la metodología aplicada a la hora de generar la descripción RTL equivalente.

## 6.6. Conclusión

En este capítulo se ha descrito el proceso de validación del sistema, indicando en primer lugar las prestaciones que resulta de interés medir para concluir si el sistema cumple con los objetivos definidos de forma previa al comienzo del proyecto, así como la metodología de medida seguida.

Posteriormente, se indican los resultados obtenidos en términos de latencia (tanto de los bloques IP *custom* de forma independiente como de la plataforma completa), *throughput* y factor de utilización de las *slices* del sistema. Para finalizar, se ha realizado una comparativa con trabajos similares que se pueden encontrar en el estado del arte de la materia, obteniendo tanto para la etapa de filtrado como para la de descifrado buenos resultados en las ratios calculadas para eliminar el factor tecnológico existente en función del dispositivo físico empleado para realizar la implementación final.

# Capítulo 7. Conclusiones y líneas futuras

## 7.1. Introducción

En este último capítulo del documento se pretende, tras haber descrito en los apartados anteriores con total detalle tanto el proceso de diseño y verificación de los bloques IP de forma individual como su integración en una plataforma que conforma el sistema completo y su posterior validación en conjunto, aportar una serie de conclusiones alcanzadas en cuanto al logro de los objetivos propuestos inicialmente se refiere. Asimismo, se aportan un conjunto de ideas o cuestiones pendientes de resolver que podrían ser tomadas como punto de partida para futuros trabajos vinculados con la materia.

## 7.2. Conclusiones del trabajo

Llegados a este punto del documento, una vez finalizada toda la parte de descripción del proceso de diseño e implementación del sistema completo y habiendo obtenido los resultados derivados en términos de PPA, se puede concluir que la solución realizada para asegurar las comunicaciones entrantes a un nodo *Fog* a través del uso de un acelerador *hardware* que agilice las etapas de filtrado y descifrado de la información cumple con su objetivo, constituyendo una solución eficiente especialmente en términos de área y consumo de energía, requisito indispensable para una aplicación orientada al Internet de las Cosas. Además, el *speedup* obtenido respecto a una descripción equivalente puramente *software* pone de manifiesto las ventajas del empleo de este tipo de dispositivos electrónicos y justifica el auge que está teniendo su empleo en la actualidad en multitud de aplicaciones vinculadas al ámbito digital.

En comparación con un *host* que realice la misma tarea, obtenemos una reducción del consumo de potencia de hasta dos órdenes de magnitud, destacando además la capacidad de este dispositivo para ampliar su funcionalidad en función de los condicionantes que puedan ir surgiendo con el transcurrir del tiempo, adaptándose a la situación actual del entorno vinculado a su aplicación final. Para ello, basta con interrumpir momentáneamente su ejecución y realizar la reconfiguración que se considere oportuna.

Asimismo, hay que señalar que gran parte del trabajo realizado se ha destinado al estudio exhaustivo de soluciones de cifrado eficientes desde un punto de vista algorítmico y de prestaciones, así como reducidas en términos de área y consumo de potencia, debido a la incapacidad de integrar AES en un sistema electrónico empotrado orientado a *low power* por el elevado consumo de energía derivado de su actividad. De esta forma, el empleo del algoritmo Simon supone un avance significativo, ya que se ha demostrado que ofrece buenos resultados con un *footprint* mínimo y un consumo de potencia reducido, especialmente si se compara con el AES.

Además, es necesario resaltar las ventajas que presenta el modelado en alto nivel frente a implementaciones realizadas en lenguajes HDL, permitiendo una mayor reusabilidad del diseño como bloque IP. De este modo, los módulos diseñados pasan a formar parte del repositorio de módulos que se ha ido conformando en la división SICAD en los últimos años (bloques de gestión de paquetes Ethernet, motores de búsqueda para arquitecturas DPI, etc.), para su reutilización en proyectos posteriores para aplicaciones de seguridad en la red.

Finalmente, me gustaría destacar el hecho de que con este trabajo se refuerza la línea de trabajo que se desarrolla en la división, sirviendo como de punto de partida para futuros proyectos que se pueda plantear realizar en un ámbito tan de actualidad y tan necesario de abordar como es la ciberseguridad. Especialmente este trabajo contribuye de forma significativa en la consecución de bloques IP reutilizables en alto nivel, ligeros en cuanto a huella de ocupación y eficientes en términos de prestaciones y de uso de recursos.

## 7.3. Trabajos futuros planteados

A pesar de haber cubierto en este proyecto las demandas de seguridad que precisan los dispositivos a incluir en el nuevo paradigma de red que es el *Fog Computing* en el sentido

de comunicación desde un *host* conectado a la nube hacia el *Fog Node*, sigue pendiente asegurar las comunicaciones con los dispositivos IoT bajo el rango de actuación del *Fog Node*. En este sentido, ya se han publicado algunos trabajos en los que se trata, especialmente, tanto la seguridad de la información transmitida en este canal de comunicación como la autenticación por parte de los dispositivos, con el fin de que no se acceda a la información albergada en el *Fog Node* mediante ataques realizados a nivel físico sobre dichos dispositivos [83].

Como estándar de comunicación, se plantea el uso de 6LoWPAN, que permite el uso de IPv6 como protocolo de red sobre redes basadas en IEEE 802.15.4, haciendo posible que los dispositivos IoT basados en tecnologías *Wireless* puedan comunicarse con otros dispositivos basados en el protocolo IP [84]; la implementación de este protocolo llevaría asociada la incorporación de algún tipo de sistema operativo empotrado en alguno de los *cores* de los que dispone el dispositivo Zynq, como puede ser FreeRTOS, o hacer uso de un *stack* TCP/IP que incluya compatibilidad con el protocolo 6LoWPAN, como puede ser µIP [85].

Asimismo, para identificar a los dispositivos hay algunos parámetros únicos y exclusivos de cada uno de ellos, como es el PUF [86], con lo que no habría lugar a dudas sobre si el dispositivo que intenta acceder al *Fog Node* es el correcto o no; de esta forma, se evitaría emplear para el proceso de autenticación parámetros de red como direcciones IP que son imitables de forma más sencilla que aquellas características propias del dispositivo en sí.

Una vez incorporada esta segunda etapa de trabajo, el *Fog Node* podría ser testeado con dispositivos IoT de campo reales, incluyendo algún tipo de adaptador (conectado a alguno de los puertos USB 2.0 disponibles en la placa de prototipado) para comunicaciones inalámbricas con el protocolo de bajo consumo a nivel de enlace que se considere establecer (Bluetooth LE o ZigBee) y verificando de esta forma la especificación completa del sistema, lo que le daría una utilidad real y que podría derivar en un interés industrial en su utilización.

Finalmente, en relación con la etapa de autenticación implementada mediante los CBF en el presente TFM, se propone realizar una adaptación para poder emplear el sistema en comunicaciones inalámbricas, ya que la cabecera de los paquetes en este caso es

significativamente diferente a la que presenta un *header* Ethernet, por lo que se precisa de un ajuste de los campos a analizar.

# Referencias


[1] T. J. Barnett, A. Sumits, S. Jain, and U. Andra, “Cisco Visual Networking Index (VNI): Global Mobile Data Traffic Forecast Update, 2016-2021,” 2015 [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html

[2] D. Evans, “The Internet of Things - How the Next Evolution of the Internet is Changing Everything,” *CISCO white Pap.*, no. April, pp. 1–11, 2011 [Online]. Available: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:The+Internet+of+Things+-+How+the+Next+Evolution+of+the+Internet+is+Changing+Everything#0

[3] EPoSS, *Internet of Things in 2020 A roadmap for the future*. 2008 [Online]. Available: http://www.smart-systems-integration.org/public/documents/publications/Internet-of-Things_in_2020_EC-EPoSS_Workshop_Report_2008_v3.pdf

[4] R. Lineback, “Long-Term Internet of Things Semiconductor Forecast, IC Insights,” *Res. Bull.*, 2017 [Online]. Available: http://www.icinsights.com/news/bulletins/LongTerm-Internet-Of-Things-Semiconductor-Forecast-Reduced/

[5] K. Adam, I. Hammad, M. Adam, I. Fakharaldien, and M. A. Majid, “Big Data Analysis and Storage,” *Int. Conf. Oper. Excell. Serv. Eng.*, pp. 648–659, 2015.

[6] M. Erol-Kantarci and H. T. Mouftah, “Energy-Efficient Information and Communication Infrastructures in the Smart Grid: A Survey on Interactions and Open Issues,” *IEEE Commun. Surv. Tutorials*, vol. 17, no. 1, pp. 179–197, 2015.

[7] H. Arasteh, V. Hosseinnezhad, V. Loia, A. Tommasetti, O. Troisi, M. Shafie-khah, and P. Siano, “IoT-based Smart Cities : a Survey,” *Environ. Electr. Eng. (EEEIC), 2016 IEEE 16th Int. Conf.*, no. August, pp. 2–7, 2016 [Online]. Available: http://ieeexplore.ieee.org/document/7555867/

[8] Synopsys Inc., “Medical Device Security : An Industry Under Attack and Unprepared to Defend,” no. May, 2017 [Online]. Available: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/medical-device-security-ponemon-synopsys.pdf

[9] A. Bannister, “The numbers behind the inexorable rise of cyber threats,” *IFSEC Global*, 2017. [Online]. Available: https://www.ifsecglobal.com/numbers-inexorable-rise-cyber-threats/

[10] Dyn Inc., “Dyn Statement on 10/21/2016 DDoS Attack.” 2016 [Online]. Available: http://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/?imm_mid=0ebbf3&cmp=em-webops-na-na-newsltr_security_20161227

[11] D. Goldman, “Chrysler recalls 1.4 million hackable cars,” *CNN tech*, 2015. [Online]. Available: http://money.cnn.com/2015/07/24/technology/chrysler-hack-recall/

[12] S. Shah, “DDoS attacks leave Finland housing without heating,” *CLoudPro*, 2016. [Online]. Available: http://www.cloudpro.co.uk/leadership/risks/6446/ddos-attacks-

leave-finland-housing-without-heating

[13] G. A. Fink, D. V. Zarzhitsky, T. E. Carroll, and E. D. Farquhar, “Security and privacy grand challenges for the Internet of Things,” *2015 Int. Conf. Collab. Technol. Syst.*, vol. 9, pp. 27–34, 2015 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7210391

[14] M. Orcutt, “Security Experts Warn Congress That the Internet of Things Could Kill People,” *MIT Technology Review*, 2016. [Online]. Available: https://www.technologyreview.com/s/603015/security-experts-warn-congress-that-the-internet-of-things-could-kill-people/

[15] Black Hat EU, “Cybercrime In The Deep Web,” 2015 [Online]. Available: https://www.blackhat.com/docs/eu-15/materials/eu-15-Balduzzi-Cybercrmine-In-The-Deep-Web-wp.pdf

[16] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog Computing and Its Role in the Internet of Things,” *Proc. first Ed. MCC Work. Mob. cloud Comput.*, pp. 13–16, 2012 [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513\npapers2://publication/doi/10.1145/2342509.2342513

[17] J. Frahim, C. Pignataro, J. Apcar, and M. Morrow, “Securing the Internet of Things: A Proposed Framework- Cisco Inc.,” *Cisco.Com*. pp. 1–7, 2016 [Online]. Available: http://www.cisco.com/c/en/us/about/security-center/secure-iot-proposed-framework.html

[18] A. V. Dastjerdl and R. Buyya, “Fog Computing : Helping the Internet of Things Realize its Potential,” *IEEE Computer Society*, 2016.

[19] B. Negash, A. M. Rahmani, P. Liljeberg, and A. Jantsch, “Fog Computing Fundamentals in The Internet-of-Things,” in *Fog Computing in the Internet of Things. Intelligence at the Edge*, 2017, pp. 1–11.

[20] M. Chiang and T. Zhang, “Fog and IoT: An Overview of Research Opportunities,” *IEEE Internet Things J.*, vol. 4662, no. c, pp. 1–1, 2016 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7498684

[21] I. Stojmenovic and S. Wen, “The Fog Computing Paradigm: Scenarios and Security Issues,” *Proc. 2014 Fed. Conf. Comput. Sci. Inf. Syst.*, vol. 2, pp. 1–8, 2014 [Online]. Available: https://fedcsis.org/proceedings/2014/drp/503.html

[22] K. Lee, D. Kim, D. Ha, U. Rajput, and H. Oh, “On security and privacy issues of fog computing supported Internet of Things environment,” *2015 Int. Conf. Netw. Futur. NOF 2015*, 2015.

[23] S. Kolluri, “Leveraging Power Leadership at 28 nm with Xilinx 7 Series FPGAs Process Technology,” vol. 436, pp. 1–16, 2015.

[24] A. Telikepalli, “Power vs . Performance : The 90 nm Inflection Point Reducing Power in FPGAs The Triple Challenge,” vol. 223, pp. 1–18, 2006.

[25] H. Chen, Y. Chen, and D. H. Summerville, “A survey on the application of FPGAs for network infrastructure security,” *IEEE Commun. Surv. Tutorials*, vol. 13, no. 4, pp. 541–561, 2011.

[26] K. Freund, “Amazon’s Xilinx FPGA Cloud: Why This May Be A Significant Milestone,” *Forbes*, 2016. [Online]. Available: http://www.forbes.com/sites/moorinsights/2016/12/13/amazons-xilinx-fpga-cloud-why-this-may-be-a-significant-milestone/#c9c78cb3fb4c

[27] C. Kachris and D. Soudris, “A Survey on Reconfigurable Accelerators for Cloud Computing,” *26th Int. Conf. F. Program. Log. Appl.*, 2016 [Online]. Available: http://www.irdindia.co.in/journal/journal_ijacte/pdf/vol3_iss1/1.pdf

[28] P. K. Gupta, “Xeon+ FPGA Platform for the Data Center,” *Fourth Work. Intersect. Comput. Archit. Reconfigurable Log.*, vol. 119, 2015.

[29] Xilinx Inc., “Zynq-7000 All Programmable SoC Overview,” vol. 190, pp. 1–21, 2016.

[30] Xilinx Inc., “Zynq ® UltraScale +™ MPSoCs,” *Prod. Guid.*, pp. 1–19, 2017.

[31] Avnet, “Zedboard ( Zynq™ Evaluation and Development ) Hardware User’s Guide,” no. January, 2014.

[32] Xilinx Inc., “Vivado Design Suite User Guide Getting Started,” vol. 901, pp. 1–120, 2013.

[33] Xilinx Inc., “Vivado Design Suite High-Level Synthesis,” vol. 907, pp. 1–672, 2016.

[34] F. Winterstein, S. Bayliss, and G. a. Constantinides, “High-level synthesis of dynamic data structures: A case study using Vivado HLS,” *2013 Int. Conf. Field-Programmable Technol.*, pp. 362–365, 2013 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6718388

[35] Xilinx Inc., “Vivado Design Flows Overview,” vol. 907, 2016.

[36] Xilinx Inc., “UltraFast Design Methodology Guide for the Vivado Design Suite (v2014.1),” vol. 949, pp. 1–322, 2014.

[37] Xilinx Inc., “An Introduction to the SDSoC Environment,” vol. 1028, pp. 1–82, 2016.

[38] Standards.ieee.org, “IEEE 802.3: Ethernet.” 2017 [Online]. Available: https://standards.ieee.org/about/get/802/802.15.html

[39] A. Rayes and S. Samer, “Internet of Things From Hype to Reality,” pp. 35–57, 2017 [Online]. Available: http://link.springer.com/10.1007/978-3-319-44860-2

[40] A. J. Jara, L. Ladid, and A. Skarmeta, “The Internet of Everything through IPv6: An Analysis of Challenges, Solutions and Opportunities,” *J. Wirel. Mob. Networks, Ubiquitous Comput. Dependable Appl.*, vol. 4, no. 3, pp. 97–118, 2013.

[41] Firewall.Cx, “IPv6 Subnetting - How and Why to Subnet IPv6,” 2017. [Online]. Available: http://www.firewall.cx/networking-topics/protocols/877-ipv6-subnetting-how-to-subnet-ipv6.html

[42] T. Gomes, S. Pinto, T. Gomes, A. Tavares, and J. Cabral, “Towards an FPGA-based edge device for the Internet of Things,” *IEEE Int. Conf. Emerg. Technol. Fact. Autom. ETFA*, vol. 2015-Octob, 2015.

[43] A. Bussa, C. Prakash, and A. Amdapurkar, “Overview of UDP/IP Protocol,” *Sion Semiconductors*, 2017. [Online]. Available: http://www.sionsemi.com/whitepapers/udp-ip-overview.html

[44] Xilinx Inc., “PetaLinux SDK User Guide Zynq All Programmable SoC Linux-FreeRTOS AMP Guide,” vol. 978, 2013.

[45] G. Tuquerres, M. R. Salvador, and R. Sprenkels, “Mobile IP:Security & Application,” 1999.

[46] Universität Innsbruck, “The IPSec Security Architecture for the Internet Protocol,” 2012 [Online]. Available: http://www.ccs-labs.org/teaching/netsec/2012s/11_IPSec.pdf

[47] S. Kent and K. Seo, “Security Architecture for the Internet Protocol,” 2005 [Online]. Available: https://tools.ietf.org/html/rfc4301

[48] M. Rao and T. Newe, “An FPGA Based Reconfigurable IPSec ESP Core suitable for IoT applications,” *10th Int. Conf. Sens. Technol.*, pp. 1–5, 2016.

[49] Y. Niu, L. Wu, L. Wang, X. Zhang, and J. Xu, “A configurable IPSec processor for high performance in-line security network processor,” *Proc. - 2011 7th Int. Conf. Comput. Intell. Secur. CIS 2011*, pp. 674–678, 2011.

[50] D. J. Bernstein, “Curve25519: New Diffie-Hellman speed records,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 3958 LNCS, pp. 207–228, 2006.

[51] OpenSSL Software Foundation, “OpenSSL Cryptography and SSL/TLS Toolkit,” *User Guid.*, pp. 1–225, 2017.

[52] D. J. Bernstein, T. Lange, and P. Schwabe, “The security impact of a new cryptographic library,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7533 LNCS, pp. 159–176, 2012.

[53] F. Denis, “Sodium Crypto Library,” *Eng. Med. Biol. Soc. 2008. EMBS 2008. 30th Annu. Int. Conf. IEEE*, vol. 24, p. 2004, 2003.

[54] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers, “TweetNaCl: A Crypto Library in 100 Tweets,” pp. 64–83, 2013.

[55] B. J. Mohd, T. Hayajneh, and A. V. Vasilakos, “A survey on lightweight block ciphers for low-resource devices: Comparative study and open issues,” *J. Netw. Comput. Appl.*, vol. 58, pp. 73–93, 2015 [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2015.09.001

[56] R. Avanzi, “A Salad of Block Ciphers.,” *IACR Cryptol. ePrint Arch.*, p. 296, 2016 [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2016.html#Avanzi16a

[57] J. Woods and P. Muoio, “Practical applications of Lightweight Block Ciphers to Secure EtherNet / IP Networks,” *ODVA Ind. Conf. 17th Annu. Meet.*, pp. 1–15, 2015.

[58] M. Appel, A. Bossert, S. Cooper, T. Kußmaul, L. Johannes, C. Pauer, and A. Wiesmaier, *Block ciphers for the IoT – SIMON , SPECK , KATAN , LED , TEA , PRESENT , and SEA compared*. 2016.

[59] E. Gulcan, A. Aysu, and P. Schaumont, “A flexible and compact hardware architecture for the SIMON block cipher,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 8898, pp. 34–50, 2015.

[60] R. Nithya and D. S. Kumar, “AES Is For the Present Era , Whereas Simon Could Be For

the Future," *Int. J. Adv. Res. Electr. Electron. Instrum. Eng.*, vol. 5, no. 4, pp. 68–74, 2016.

[61] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "Simon and Speck: Block Ciphers for the Internet of Things," 2015 [Online]. Available: http://csrc.nist.gov/groups/ST/lwc-workshop2015/papers/session1-shors-paper.pdf

[62] J. Zhu and N. Dutt, "Electronic system-level design and high-level synthesis," in *Electronic Design Automation*, 2009, pp. 235–298.

[63] P. P. Carballo, "Aportaciones a la metodología de diseño basada en síntesis de alto nivel. Aplicaciones al diseño de IPs para procesado de eventos complejos y codificación de vídeo," 2016.

[64] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, "An Introduction to High-Level Synthesis," pp. 8–17, 2009.

[65] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, 2011.

[66] Xilinx Inc., "Python productivity for Zynq (Pynq) Documentation," 2017 [Online]. Available: https://media.readthedocs.org/pdf/pynq/latest/pynq.pdf

[67] L. Maillet-Contoz and F. Ghenassia, "Transaction Level Modeling. An Abstraction Beyond RTL," in *Transaction Level Modeling with SystemC*, 2005, pp. 23–55.

[68] Y. B. Alfaro, "Implementación de búsquedas regulares en una plataforma para el procesamiento de paquetes de red en FPGA Xilinx Zynq," 2016 [Online]. Available: http://hdl.handle.net/10553/21108

[69] R. Quinnell, "Embedded Markets Study: Changes in Today's Design, Development & Processing Environments," *ARM TechCon*, 2015.

[70] Cadence Inc., "Stratus High-Level Synthesis," pp. 1–3, 2016 [Online]. Available: http://www.europractice.stfc.ac.uk/vendors/cadence_stratus_ds.pdf

[71] Synopsys Inc., "Synplify Pro and Premier," 2015 [Online]. Available: https://www.synopsys.com/Tools/Implementation/FPGAImplementation/CapsuleModule/synplify-pro-premier.pdf

[72] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and Practice of Bloom Filters for Distributed Systems," *IEEE Commun. Surv. Tutorials*, vol. 14, no. 1, pp. 131–155, 2012 [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5751342

[73] S. Dharmapurikar, M. Attig, and J. Lockwood, "Design and Implementation of a String Matching System for Network Intrusion Detection using FPGA-based Bloom Filters," *Proc. 12 th Annu. IEEE Symp. FieldProgrammable Cust. Comput. Mach.*, vol. 1, no. 3, pp. 186–189, 2004.

[74] M. Ahmadi, "Hashing Functions Performance in Packet Classification," *Proc. Int. Conf.*, 2007 [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.62.9642&rep=rep1&type=pdf

[75] J. Seward, N. Nethercote, T. Hughes, J. Fitzhardinge, J. Weidendorfer, and P. Mackerras, “Valgrind documentation,” 2017.

[76] Kmarquet, “GitHub - Source code developed in the BLOC project.” [Online]. Available: https://github.com/kmarquet/bloc

[77] J. Loinig, J. Wolkerstorfer, and A. Szekely, “Packet Filtering in Gigabit Networks Using FPGAs,” pp. 1–8, 2007 [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.7016&rep=rep1&type=pdf

[78] J. Rohrbeck, V. Altmann, S. Pfeiffer, D. Timmermann, M. Ninnemann, and R. Maik, “Secure Access Node : an FPGA-based Security Architecture for Access Networks,” *ICIMP 2011 Sixth Int. Conf. Internet Monit. Prot. Secur.*, pp. 54–57, 2011.

[79] H. Yu, R. Cong, L. Chen, and Z. Lei, “Blocking pornographic, illegal websites by internet host domain using FPGA and bloom filter,” *Proc. - 2010 2nd IEEE Int. Conf. Netw. Infrastruct. Digit. Content, IC-NIDC 2010*, pp. 619–623, 2010.

[80] A. Aysu, E. Gulcan, and P. Schaumont, “SIMON says: Break area records of block ciphers on FPGAs,” *IEEE Embed. Syst. Lett.*, vol. 6, no. 2, pp. 37–40, 2014.

[81] J. Wetzels and W. Bokslag, “Simple SIMON FPGA implementations of the SIMON64/128 Block Cipher,” *arXiv Prepr. arXiv1507.06368*, 2015.

[82] Y. Jararweh, L. Tawalbeh, H. Tawalbeh, and A. Moh’d, “28 Nanometers FPGAs Support for High Throughput and Low Power Cryptographic Applications,” *J. Adv. Inf. Technol.*, vol. 4, no. 2, pp. 84–90, 2013.

[83] M. Saadeh, A. Sleit, M. Qatawneh, and W. Almobaideen, “Authentication techniques for the internet of things: A survey,” *Proc. - 2016 Cybersecurity Cyberforensics Conf. CCC 2016*, pp. 28–34, 2016.

[84] C. Hennebert and J. Dos Santos, “Security protocols and privacy issues into 6LoWPAN stack: A synthesis,” *IEEE Internet Things J.*, vol. 1, no. 5, pp. 384–398, 2014.

[85] Y. Chen, K. M. Hou, H. Zhou, H. L. Shi, X. Liu, X. Diao, H. Ding, J. J. Li, and C. De Vaulx, “6LoWPAN stacks: A survey,” *7th Int. Conf. Wirel. Commun. Netw. Mob. Comput. WiCOM 2011*, pp. 6–9, 2011.

[86] H. Akhundov, “MSc THESIS Design & development of public-key based authentication architecture for IoT devices using,” no. 1, 2017.