UNIVERSIDAD DE SEVILLA

TESIS DOCTORAL

LILA : Low-level IoT Ligtweigth Applications: Aplicaciones Hardware para dispositivos IoT

Autor:
Germán Cano Quiveu

Director:
Manuel Jesús Bellido Díaz

Director:
Paulino Ruíz de Clavijo Vázquez

Investigación y Desarrollo Digital (ID2)
Departamento de Tecnología Electrónica

11 de marzo de 2022

Tesis doctoral subvencionada por el Ministerio de Economía y Competitividad del Gobierno de España bajo el Proyecto de Investigación TIN2017-89951-P: *BootTimeIoT: Sistemas de inicio avanzados y sincronización temporal de alta precisión para IoT*.

UNIVERSIDAD DE SEVILLA

Resumen

Escuela Técnica Superior de Ingeniería Informática

Departamento de Tecnología Electrónica

Doctor en Ingeniería Informática

LILA : Low-level IoT Ligtweigth Applications: Aplicaciones Hardware para dispositivos IoT

por Germán Cano Quiveu

Esta tesis se enmarca dentro del Proyecto de Investigación TIN2017-89951-P: *BootTimeIoT: Sistemas de inicio avanzados y sincronización temporal de alta precisión para IoT*, específicamente dentro de la línea del proyecto dedicada a «Bootloaders y Sistemas de archivos para sistemas empotrados». La tesis se presenta como compendio de publicaciones y esta constituida de tres partes diferenciadas. En la primera de ellas se presenta una aplicación de bootloader hardware para dispositivos IoT, que es uno de los objetivos/tarea principales del proyecto de investigación. En el desarrollo de esta tarea surgen dos cuestiones importantes relacionadas con el diseño de aplicaciones IoT y que dan lugar a las otras dos partes de esta tesis: Una metodología integral universal para Aplicaciones sobre FPGA, la cual permite una verificación mediante simulación RTL y On-Chip; y un Core que aporta seguridad a los dispositivos IoT confiriendo confidencialidad, integridad y autenticidad a los datos de usuario haciendo uso de esta metodología integral.

Agradecimientos

Quiero agradecer esta tesis a las personas que forman el grupo de investigación ID2, los cuales me han ayudado no solo en la realización de esta tesis así como en labores docentes, sino en aprender a vivir con una filosofía de vida mucho mas adecuada tanto física como mentalmente. Además de brindarme apoyo en ambas facetas en todo momento. Destacar a mis tutores de tesis, Manolo y Paulino, pues han sido sin duda los que más han contribuido en todos estos aspectos, la tesis, enseñarme como afrontar una clase, y en especial a como vivir más saludablemente sin estar pensando en todas y cada una de las consecuencias y por tanto nunca hacer nada o estar preocupado todo el tiempo, por todo esto siempre les estaré agradecido.

Quiero destacar que yo he llegado a este punto por el constante apoyo y cariño familiar que he recibido durante toda mi vida, por tanto, se podría decir que han sido coautores de esta tesis. Destacar en especial a mis tíos Rafa y Chiruca, que siempre han estado ahí cerca de nosotros en todo momento. Mi prima Anuska que me hizo ver la importancia de la salud y de como afrontar la vida con optimismo. Dani, el cual me llevo a tomar el camino de la informática. Mi hermana Maite, cuyo desparpajo y complicidad han hecho de este camino uno mucho mas entretenido. Mis padres Gerardo y Loli, los cuales me han educado y han hecho siempre lo posible para que yo pudiera continuar estudiando sin tener que preocuparme por ninguna otra cosa. Por último, quiero destacar a dos personas. Mi prima Chiqui, pues aparte de ser mi madrina, es la persona que hizo que me quisiera dedicar a la investigación y a no rendirme en mi empeño. La otra persona es aquella que ha tenido el mayor impacto en mi vida. Mi abuela Emilia (aunque no le gustase ese nombre y siempre usáramos otro para referirnos a ella) quien, desde que tengo memoria, ha estado con nosotros, criandonos y ante todo mostrando un cariño incondicional, por tanto, puedo asegurar que todo lo que he conseguido en mi vida se lo debo a ella, siempre fue mi referente así como el de toda la familia. Es por ello que esta tesis esta dedicada especialmente a mi abuela Lila.

Índice general

Resumen III

Agradecimientos V

Índice general VII

Índice de figuras IX

Índice de tablas XI

Abreviaturas XIII

1. Introducción y Objetivos de la tesis 1
- 1.1. Introducción . . . 1
- 1.2. Objetivos de la tesis . . . 3
- 1.3. Estructura de la tesis doctoral . . . 3

2. Verificación funcional: Introducción 5
- 2.1. Introducción . . . 5
- 2.2. Conceptos . . . 5
  - 2.2.1. *Unit Under Test (UUT)* . . . 5
  - 2.2.2. *Black-box testing* y *White-box testing* . . . 5
  - 2.2.3. Verificación exhaustiva y no exhaustiva . . . 6
    - *Test Pattern Generator* (TPG) . . . 6
  - 2.2.4. Verificación On-chip y Simulación RTL . . . 7
    - Analizadores Lógicos . . . 7
    - *Built-In Self Test* (BIST) . . . 8

3. Criptografía Aplicada: Introducción 9
- 3.1. Introducción . . . 9
- 3.2. *Stream Ciphers* . . . 11
- 3.3. *Block Ciphers* . . . 12
  - 3.3.1. Modos de Operación . . . 14
    - ECB . . . 15
    - CTR . . . 15
- 3.4. Funciones *hash* . . . 15
  - 3.4.1. Clasificación de funciones *hash* . . . 16
  - 3.4.2. *Preimage Resistance* . . . 17
  - 3.4.3. *Second Preimage Resistance* . . . 17
  - 3.4.4. *Collision Resistance* . . . 17

3.5. *MAC* 19
3.5.1. *HMAC* 19
3.6. *Key Derivation Function* 22

4. Resumen global de los resultados **23**
4.1. Bootloader Hardware 23
4.2. Metodología de verificación de IPCores 24
4.2.1. Definición Conceptual de la metodología 24
4.2.2. Modulo Autotest Core 25
4.2.3. Resultados 26
4.3. *Embedded LUKS* 31
4.3.1. Introducción 31
4.3.2. Diseño de E-LUKS 31
4.3.3. Resultados 34

5. Publicaciones **39**
5.1. An Integrated Digital System Design Framework With On-Chip Functional Verification and Performance Evaluation . . . 39
5.1.1. Breve resumen 39
5.1.2. Datos Revista 39
5.2. Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security 53
5.2.1. Breve resumen 53
5.2.2. Datos Revista 53
5.3. Address-encoder byte order 76
5.3.1. Breve resumen 76
5.3.2. Datos Revista 76
5.4. OpenRISC hardware bootloader over WiFi 86
5.4.1. Breve resumen 86
5.4.2. Datos Conferencia 86

6. Conclusiones y Trabajo futuro **87**
6.1. Conclusiones 87
6.2. Trabajo Futuro 89

Bibliografía: Artículos derivados de la tesis **91**

Bibliografía **93**

Índice de figuras

3.1. diagrama de un *stream cipher* sicrono/asíncrono 12
3.2. *SP-network* 14
3.3. esquema del *CTR mode* 16
3.4. construccion *HMAC* 21

4.1. Representación de la metodología propuesta 25
4.2. Esquemático del Autotest Core 26
4.3. Tiempo medio de ejecución de un patrón para el Core PRESENT en escala log10. 29
4.4. Tiempo medio de ejecución de un patrón para el Core SPONGENT-88 con entrada de 1024 bytes en escala log10. 30
4.5. Tiempo de ejecución del total de patrones en el IPCore PRESENT. 30
4.6. Tiempo de ejecución del total de patrones en el IPCore SPONGENT-88 con entrada de 1024 bytes. 30
4.7. Estructura interna de E-LUKS. 32
4.8. Esquemático del Core E-LUKS. 35

Índice de tablas

4.1. Recursos tomados por diferentes herramientas de verificación funcional para el testado del *block cipher* PRESENT en la FPGA Xilinx Artix7 XC7A100T-1CSG324C. 28
4.2. Recursos tomados por diferentes herramientas de verificación funcional para el testado de la función *hash* SPONGENT-88 en la FPGA Xilinx Artix7 XC7A100T-1CSG324C. 28
4.3. Campos de la cabecera E-LUKS. 32
4.4. Campos de E-LUKS KS_x. 32
4.5. comparativa de LUKS con E-LUKS. 34
4.6. Tiempos de ejecución para la XC7A100T FPGA. 36
4.7. Comparación de recursos entre LUKS y E-LUKS para la XC7Z020 FPGA. 36
4.8. Comparativa de recursos para soluciones específicas de IoT. Los valores - no aparecen en los artículos originales. 37
4.9. Comparativa entre soluciones IoT. 37

Abreviaturas

BIST	**B**uilt-**I**n **S**elf **T**est
CBC	**C**ipher **B**lock **C**hainig
CUT	**C**ircuit **U**nder **T**est
CUT	**C**ore **U**nder **T**est
CFB	**C**ipher **F**eed**B**ack
CTR	**C**oun**T**e**R**
DDR	**D**ouble **Data Rate**
DEC_k	**DEC**rypt with key k
DUT	**D**esign **U**nder **T**est
E-LUKS	**E**mbedded-**LUKS**
ECB	**E**lectronic **C**ode **B**ook
ENC_k	**ENC**rypt with key k
FDE	**F**ull **D**isk **E**ncryption
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
I/O	**I**nput/**O**utput
IoT	**I**nternet **o**f **T**hings
IPCore	**I**ntellectual **P**ropierty **Core**
IPCUT	**IPC**ore **U**nder **T**est
IV	**I**nitial **V**alue
ILA	**I**ntegrated **L**ogic **A**nalyzer
ISA	**I**nstruction **S**et **A**rquitecture
KDF	**K**ey **D**erivation **F**unction
LUKS	**L**inux **U**nified **K**ey **S**ystem
MAC	**M**essage **A**uthentication **C**ode
MCMM	**M**ixed **M**ode **C**lock **M**anager
MIG	**M**emory **I**nterface **G**enerator
MSB	**M**ost **S**ignificant **B**its
OFB	**O**utput **F**eed**B**ack
ORA	**O**utput **R**esponse **A**nalyzer
OTA	**O**ver **T**he **A**ir
PLL	**P**hase **L**ocked **L**oop
RISC	**R**educed **I**nstruction **S**et **C**omputer
RTL	**R**egister **T**ransfer **L**evel
S-box	**S**ubstitution-**box**
SoC	**S**ystem **o**n **C**hip
SP-network	**S**ubstitution**P**ermutation-**network**

SPI	**S**erial **P**eripheral **I**nterface
SSL	**S**ingle **S**tuck **L**ine
TPG	**T**est **P**attern **G**enerator
UUT	**U**nit **U**nder **T**est
VHDL	**V**HSIC **HDL**
VIO	**V**irtual **I**nput/**O**utput

Capítulo 1

Introducción y Objetivos de la tesis

1.1. Introducción

El sector dedicado al Internet de las cosas (*Intenet Of Things, IoT*), ha experimentado un auge en los últimos años y, en su estado actual, se podría afirmar que seguirá al alza en los años venideros. Esto ha conllevado, a un extensivo uso de lógica reprogramable, tal y como podrían ser los chips FPGA, para el desarrollo de aplicaciones IoT. Los chips FPGA se han venido utilizando para el diseño y testeo de aplicaciones hardware especificas (Cores). Su gran versatilidad, proporcionada por su capacidad de reprogramación, ha hecho posible testar en un solo chip diversos Cores, ya sean a la vez o de manera secuencial. El uso extensivo de estos chips ha conllevado a que se integren en plataformas que cuentan con un amplio numero de periféricos, además de la implementación de herramientas para la programación pudiendo así facilitar la tarea de diseño. Por tanto, no es de extrañar su fama dentro del diseño de sistemas IoT. De hecho, estudios de mercado actuales apuntan a que el valor asociado a los campos del IoT y FPGA crecerán hasta 1.11 trillones de dolares en 2028 para IoT y 18.8 billones de dólares para FPGAs, teniendo un crecimiento anual del 22.8 % [Iot] y 9.8 % [Fpg] respectivamente.

Sin embargo, esta razón de crecimiento ha llevado a la producción de un gran numero de aplicaciones IoT desarrolladas sobre FPGA, con el consecuente estrés regido por la demanda del mercado. Debido a este estrés se estima que en 2018 solo el 16 % de los proyectos que salieron al mercado están libres de fallos y el 64 % del total de proyectos iban con retraso [Fos18]. Esto implica que en un mercado tan activo y agresivo como es el IoT contar con una metodología de trabajo eficaz y que concluya el correcto funcionamiento del diseño es fundamental.

Otro aspecto clave relacionado con este incremento de dispositivos IoT es la seguridad de los mismos. La cantidad de información que recae en estos dispositivos debe estar restringida sólo a usuarios/dispositivos verificados. Esto es de especial interés debido al incremento del uso de estos dispositivos en nuestra vida cotidiana, conteniendo muchos de ellos información confidencial de la persona. Se podría considerar que aquellos dispositivos

en el mercado son seguros, pero eso dista mucho de la realidad [Men+19] [Shw+18].

A pesar de estos factores, existe una directriz común para todo diseño IoT. Esta directriz se centra en utilizar la mínima cantidad de recursos posibles. Esto es debido al hecho de su manufacturación y la tendencia de aplicar IoT en cualquier ámbito. Por tanto, este punto ha sido fundamental y pieza angular para la realización de esta tesis.

Enmarcado dentro del proyecto de investigación TIN2017-89951-P: *BootTimeIoT: Sistemas de inicio avanzados y sincronización temporal de alta precisión para IoT*, esta tesis comienza desarrollando una aplicación de bootloader, completamente hardware, para dispositivos IoT. Este bootloader se integra sobre el SoC OpenRISC, implementado en una FPGA, y siendo incluso capaz de cargar el kernel de Linux sobre el mismo. Este trabajo ha sido presentado en dos publicaciones: una conferencia, la cual introduce una primera versión de este bootloader [CQ18] y un artículo en revista, este último hace uso de una versión posterior, la cual es utilizada como setup para realizar una modificación sobre el procesador y aplicar los resultados sobre un sistema operativo [GM+20].

En el desarrollo de este bootloader se observa la necesidad de disponer de una metodología integral de verificación para el desarrollo de Cores sobre FPGAs. De esta necesidad surge el primero de los trabajos principales de esta tesis con el desarrollo de una nueva propuesta de metodología de verificación integral publicada como artículo en revista [CQ+21a].

Por otra parte, en la aplicación del bootloader se observa la necesidad de dotar de seguridad al sistema IoT. De aquí nace el segundo de los trabajos principales de la tesis con el desarrollo de un Core que aporta seguridad a los datos de usuario diseñado específicamente para dispositivos IoT de bajos recursos. Este trabajo al igual que la metodología ha sido publicado como artículo en revista [CQ+21b].

Respecto a la metodología integral, los resultados muestran que agiliza el proceso de diseño y permite llevar a cabo una verificación funcional, tanto a nivel RTL como On-Chip, en unos tiempos razonables. Además, ofrece una gran flexibilidad permitiendo el uso de diversas FPGAs, debido a su diseño *Open-Source*, y el uso de diferentes estrategias de verificación al poder adaptarse a una gran variedad de patrones de test. Esta metodología se puesto a prueba en la realización del segundo Core, pieza central de la segunda publicación. Este Core aporta seguridad a los dispositivos IoT, consiguiendo proporcionar a los datos de usuario las propiedades de confidencialidad, integridad y autenticidad, dicho Core se ha denominado Embedded LUKS (E-LUKS), y al igual que la metodología integral es *Open-Source* por lo que puede ser implementado en la mayoría de dispositivos IoT. Es importante mencionar que debido a su diseño esta inclusión no requiere de grandes modificaciones en el sistema.

1.2. Objetivos de la tesis

El objetivo principal de esta tesis es la tarea T3.3 del proyecto de investigación TIN2017-89951-P: *BootTimeIoT: Sistemas de inicio avanzados y sincronización temporal de alta precisión para IoT* que dice así:

> Aplicar el nuevo sistema de ficheros para aplicaciones de bootloader para diversos sistemas empotrados, desde microcontroladores hasta SoCs complejos como OpenRISC o LowRISC

Para cubrir adecuadamente esta tarea han surgido un conjunto de objetivos específicos que se listan a continuación:

- Realización de una plataforma Hardware/Software implementable sobre FPGA que incluya un bootloader sobre el SoC OpenRISC.
- Desarrollo de una metodología de verificación de Cores de alto rendimiento sobre FPGAs que permitan:
  - Testar un diseño sobre cualquier FPGA.
  - Obtener medidas del rendimiento del Core en tiempos razonables.
- Dotar al sistema, compuesto por el bootloader más SoC, de seguridad. Para ello se proponen los siguientes aspectos:
  - Propuesta de un sistema seguro capaz de proporcionar confidencialidad, integridad y autenticidad, utilizando para ello algoritmos adecuados para dispositivos IoT de bajos recursos.
  - Diseño de un Core para que el sistema seguro propuesto pueda integrarse con dispositivos IoT de bajos recursos, sin que ello requiera de modificaciones sobre otros componentes del diseño completo.

1.3. Estructura de la tesis doctoral

Esta tesis doctoral, presentada bajo el formato de compendio de publicaciones, esta estructurada tal y como se describe a continuación:

- Capítulo 1, Introducción y Objetivos de la tesis : En este capítulo se pone en contexto la temática de la tesis así como la interrelación de las publicaciones y como estas están ligadas a un fin. Además, se recopilan los objetivos que se pretenden alcanzar en el desarrollo de esta tesis.
- Capítulo 2, Verificación Funcional (Introducción) : Este capítulo introduce conceptos sobre la verificación funcional para Cores, esta centrado en el desarrollo sobre FPGA, la intencionalidad del mismo es para presentar conceptos utilizados en la primera de las aportaciones principales.

- Capítulo 3, Criptografía Aplicada (Introducción) : Este capítulo introduce los conceptos principales sobre criptografía, generando así una preparación previa para la segunda de las aportaciones principales.
- Capítulo 4, Resumen global de los resultados : Este capítulo, recopila de forma escueta la información mas relevante y novedosa que presentan las publicaciones aportadas a esta tesis.
- Capítulo 5, Publicaciones : Este capítulo recopila las publicaciones que constituyen esta tesis doctoral.
- Capítulo 6, Conclusiones y Trabajo futuro : En este cierre de tesis, se presentan las conclusiones extraídas a lo largo de la tesis así como las líneas de trabajo futuro que abren las implementaciones mostradas en este documento.

Capítulo 2

Verificación funcional: Introducción

2.1. Introducción

Durante el desarrollo de aplicaciones sobre FPGA, un proceso, el cual debería ser primordial, es el de verificación. Este proceso toma en torno al 40 % del tiempo total destinado al proyecto. Aun así, durante 2018 solo el 16 % de todos los proyectos sobre FPGA salieron a producción sin un solo fallo [Fos18]. Por tanto no es de extrañar el auge de herramientas y metodologías para crear soluciones de verificación más eficientes.

Teniendo en cuenta que en esta tesis se presenta una metodología de verificación, es relevante conocer una serie de conceptos, los cuales serán utilizados en el artículo en el que la metodología es presentada.

2.2. Conceptos

2.2.1. *Unit Under Test (UUT)*

A la implementación que se somete a la verificación se le suele denominar *Unit Under Test* (UUT), existen variantes a este nombre como son *Desing Under Test* (DUT) o *Circuit Under Test* (CUT), entre otros.

2.2.2. *Black-box testing* y *White-box testing*

Comenzaremos por definir la manera en que se tratará al UUT. Para ello tomaremos como referencia las definiciones mostradas en [ND12]. Se define *Black-box testing* como aquella técnica que sólo tiene en cuenta las entradas y salidas del UUT, dando por tanto una casuística del comportamiento del UUT. Sin embargo, para ahondar en las señales internas del UUT se requiere

de la técnica *White-box testing*, la cual habilita la monitorización de dichas señales, pudiendo así obtener un entendimiento del UUT a lo largo de todo el proceso de ejecución.

2.2.3. Verificación exhaustiva y no exhaustiva

A continuación nos centraremos en la naturaleza de los patrones de entradas aplicados al UUT. Se dice que la verificación es exhaustiva sí y sólo sí se cubren todos los posibles patrones de entrada, dando lugar a reconocer todos los posibles resultados y por tanto pudiendo observar cualquier fallo del sistema, obteniendo así un 100 % de cubrimiento. No obstante, esto resulta inviable para determinados UUT, como puede ser un *block cipher* de 64-bit de entrada y 80-bit de clave. Ya que harían falta 2^{64+80} patrones para tener una verificación exhaustiva. En estos casos se recurre a un reducido grupo de patrones que mediante modelos matemáticos pueden obtener un alto porcentaje de cubrimiento.

En ambos supuestos, los patrones de test serán obtenidos a través del denominado *Test Pattern Generator*(TPG).

Test Pattern Generator (TPG)

Un *Test Pattern Generator* (TPG) es una metodología la cual utiliza un modelo matemático como base para obtener un número de patrones pero obteniendo con ellos un alto porcentaje de cubrimiento del UUT. El máximo cubrimiento sólo podría ser obtenido por un generación exhaustiva, algo no siempre posible por lo que los modelos matemáticos suelen ser más plausibles. Uno de los modelos más utilizado es el *Single Stuck-at Line*. Este modelo se basa en la premisa de que sólo una línea, concurrentemente, puede tener su valor fijado a 0 o 1.

Otro modelo utilizado principalmente para Cores criptográficos, sería un conjunto de patrones aleatorios, el tamaño de este conjunto vendría dado por el *coupon collector problem* [Shi07]. La forma de aplicarlo a los cores criptográficos viene dada por [DN+09], donde se define que en estos cores el componente clave a testar serían las *S-boxes*. Con esto determina que hacen falta k patrones para realizar un testado exhaustivo de la *S-box*, por lo tanto, al estar las entradas de la *S-box* uniformemente distribuidas, cada patrón aleatorio generado posee una probabilidad $p = 1/k$ de ocurrencia para las entradas de la *S-box*. Con todo esto harán falta n patrones para obtener un cubrimiento P del Core, el cálculo de n es obtenido mediante:

$$P[X \leq n] = 1 - \sum_{i=1}^{k} -1^{i+1} \binom{k}{i} (1 - ip)^n$$

2.2.4. Verificación On-chip y Simulación RTL

Por último catalogaremos la verificación según como se ejecute ésta. Si la verificación se realiza sobre el dispositivo físico donde esta implementado el UUT, se denomina Verificación On-Chip. Si, por el contrario, trabajamos en un entorno completamente software donde tenemos la implementación del UUT en un nivel RTL (*Register Transfer Level*), como pueden ser los lenguajes HDL, entonces la verificación se denomina Simulación RTL.

Respecto a la Simulación RTL, esta verificación tiene la ventaja de ser más flexible cuando se refiere a poder reutilizar librerías o lenguajes de alto nivel, véase Python en el caso de CocoTB [Coc], siendo una excelente opción para poder realizar una verificación *white-box testing*. Sin embargo, como contrapunto negativo, estos resultados pueden obviar fallos que ocurren en implementaciones hardware, como por ejemplo el *clock seer*, el retardo de señales, las variaciones producidas por el entorno, etc . . . .

En el otro lado del espectro la verificación On-Chip provee de unos resultados más fiables, al estar estos testados sobre el dispositivo físico. Sin embargo realizar una verificación *white-box testing*, además de resultar costoso, puede incurrir en un alto consumo de recursos sobre la placa. Por lo que este tipo de verificación suele presentarse junto con una verificación *black-box testing*. Existen herramientas y arquitecturas para realizar este tipo de verificación, a continuación presentaremos dos de ellas, los Analizadores Lógicos y la arquitectura *Built-In Self Test* BIST.

Analizadores Lógicos

Los Analizadores Lógicos son herramientas que permiten la lectura de una amplia cantidad de señales digitales. Por tanto, para poder realizar esta lectura necesitamos que el dispositivo físico tenga estas señales accesibles hacia el exterior. Este punto resulta un gran impedimento en dispositivos como las FPGAs donde el consumo de pines I/O es elevado si quisiéramos llevar a cabo una verificación *white-box testing*. Otro punto a tener en cuenta de estas herramientas es la cantidad de muestras que pueden capturar, y a la frecuencia máxima que se pueden obtener, para su posterior análisis.

Estas razones, además de su elevado coste, llevaron a crear Analizadores Lógicos que se implementaban dentro del propio dispositivo junto al UUT. Estos se denominan *Integrated Logic Analyzer* (ILA). Las ILAs solucionan el consumo de pines de I/O pero su limitación aparece en la cantidad de muestras que pueden capturar, ya que estas están directamente relacionada con el consumo de recursos de bloques de memoria del dispositivo.

***Built-In Self Test* (BIST)**

Built-In Self Test es una arquitectura la cual permite una verificación On-Chip. Para ello, BIST se implementa junto con el UUT en el mismo dispositivo. Su funcionalidad permite verificar el UUT durante todo su ciclo de vida. Respecto a su estructura, usaremos como referencia la mostrada en [Str]. Una estructura básica de BIST se compone de:

- Un módulo TPG el cual genera los patrones necesarios para testar el UUT.
- Un módulo para comprobar el resultado esperado de los patrones con el generado por el UUT denominado *Output Response Analyzer* (ORA).
- Un módulo que multiplexe las entradas del UUT para poner administrar los patrones provenientes del TPG, además de las señales provenientes del exterior.
- Un módulo de control capaz de controlar los módulos anteriores para realizar correctamente la verificación BIST.

Capítulo 3

Criptografía Aplicada: Introducción

3.1. Introducción

La información es quizás uno de los recursos más importantes, es lógico, por tanto, que la escritura fuera desarrollada por todas las civilizaciones, indistintamente de su localización geográfica, como forma de recoger el conocimiento u otra información. Esto conllevó a la necesidad de que un mensaje no fuera interceptado o tergiversado, siendo éste un punto de interés para el ser humano desde casi sus comienzos.

Si nos remontamos a la cuna de la civilización, Mesopotamia, la región entre dos ríos, el Tigris y el Eufrates, observamos como el hecho de ocultar o verificar la información era algo habitual. Se han descubierto tablillas de arcilla del 1500 a.c. donde se utiliza la escritura cuneiforme, en el que se encuentra una fórmula de cerámica vidriada que esta cifrada de tal forma que ésta quedara en secreto y no pudiera replicarse por ningún otro artesano [Kah96]. También se conoce que desde los sumerios (3300 a.c) era muy habitual el poseer sellos cilíndricos con los cuales el poseedor podía autentificar un mensaje, ya que estos sellos eran personales y distintos entre ellos para poder identificar al poseedor [Ber03]. Por último se han hallado restos arqueológicos que nos demuestran que la civilización mesopotámica utilizaba «sobres» de arcilla en el cual grababan su sello personal, dando a entender de esta manera que si el sobre estaba roto el mensaje podía haber sido no solo leído sino que incluso modificado por otra persona [Env].

Posteriormente se sofistico el arte de esconder un mensaje a simple vista para el enemigo, ejemplos de estos podrían ser el Scytale utilizado por los espartanos o el sistema desarrollado por el griego Polybius [Kah96]. Sin embargo en esta etapa clásica hay un cifrado que sobresale del resto y ha pasado a la posteridad con nombre propio, es el cifrado césar. Este cifrado utilizado por Julio César, usa un cifrado de sustitución. El original utilizaba un desplazamiento de tres letras en el mensaje, codificando así la letra A por la letra D y la letra Z por la C. Este cifrado, independientemente del desplazamiento elegido, ha sido denominado cifrado cesar. Es cierto que es vulnerable ya que en un

alfabeto de 26 caracteres existen sólo 25 posibles combinaciones por lo que es fácilmente descifrable. El cifrado de sustitución evolucionó con el tiempo haciendo que cada letra pudiera sustituirse por cualquier otra del alfabeto, creando así 26! combinaciones. Sin embargo, estos cifrados seguirían siendo vulnerables a los denominados ataques de diccionario, ya que las propiedades estadísticas del texto original serían las mismas que las del texto cifrado, como ya reflejaba Ibn ad-Duraihim en el siglo XIV [Kah96].

Hay un capítulo en la historia que está estrechamente relacionado tanto con la criptografía como con la criptología. Este hecho fue descifrar la máquina Enigma. Enigma fue utilizada durante la Segunda Guerra Mundial por las fuerzas del Eje y descifrada por los Aliados logrando con ello una ventaja estratégica que, junto con otro factores, llevo a la victoria. Esto demostró la importancia de la información y puso en valor a la criptología.

En la actualidad, Internet nos ha dado la globalización y por ende ha posibilitado el acceso de conocimiento a cualquier área. Esta globalización ha conseguido que los avances en las distintas ramas de la ciencia sean cada vez, no solo prolíficos, sino impactantes y argumentados, al poder ser expuestos los argumentos de forma inmediata a cualquier experto sin importar su localización. Esto también ha llevado a un nuevo estilo de vida, mucho más acomodada y al poder obtener alimentos y otros servicios de primera necesidad, así como artículos de ocio desde la comodidad del hogar. Sin embargo, como contrapartida, todos estos nuevos canales de comunicación han sido nuevas plataformas para que entidades no deseadas pudieran obtener información y tergiversarla para su propio beneficio. Por ello, en la actualidad, donde la información es sin lugar a duda el recurso más valioso que existe, la criptología forma parte intrínseca para asegurar la privacidad de la persona, y ésta no para de cambiar para adaptarse a los nuevos y continuos ataques que aparecen.

Nos hemos referido en varias ocasiones a la criptología pero sin definir que es. Comenzaremos con su origen etimológico. Criptología se compone de las palabras griegas *kriptos*, que significa oculto, y de *logia* cuyo significado es estudio o ciencia. Es, por tanto, el estudio de ocultar información hacia entidades no deseadas. Existen dos ramas: La criptografía, que tiene como objetivo el ocultar la información y el criptoanálisis, la ciencia que estudia las posibles vulnerabilidades de los distintos algoritmos criptográficos [PP10].

Dentro de la criptografía se encuentran tres vertientes :

- Cifrados simétricos: son aquellos algoritmos de encriptación en los que las dos partes de la comunicación requieren de la misma clave.
- Cifrados asimétricos: estos algoritmos de encriptación hacen uso de claves públicas y privadas que son distintas para cada parte. Su función habitualmente es ser parte integral de las aplicaciones de firmas digitales o intercambio de claves de forma segura para utilizar cifrados simétricos.

- Protocolos criptográficos: en esta categoría se incluyen el resto de algoritmos criptográficos, tales como las firmas digitales, funciones *hash*, funciones MAC y funciones KDF.

Cabe recalcar que una idea muy extendida en la criptografía es la seguridad basada en ocultar los algoritmos de encriptación y desencriptación. Pero tal y como estableció Kerckhoff en 1883 [PP10]:

> Un criptosistema debería ser seguro incluso si el atacante conoce todos los detalles sobre el sistema, con la excepción de la clave. Particularmente, el sistema debería ser seguro incluso cuando el atacante conoce los algoritmos de encriptación y desencriptación.

Es importante que los algoritmos sean públicos, ya que así la comunidad de criptólogos podrá analizarlos e intentar por todos los medios encontrar vulnerabilidades. Por ello, estos sistemas públicos presentados con una cierta cantidad de tiempo y que aún no han sido explotados representan los sistemas mas fiables. Un ejemplo de seguridad por ocultación sería el sistema de protección de los DVD (CSS). Este sistema fue fácilmente atacado mediante ingeniería inversa, haciendo que este medio no generara los ingresos esperados debido a las copias ilegales que eran fácilmente reproducibles.

Este capítulo está fundamentalmente basado en dos libros de referencia, los cuales me han servido para un mejor entendimiento de este campo, estos libros son [PP10] y [KR11].

Por último, destacar que la intencionalidad de este capítulo es el de explicar aquellos conceptos requeridos a lo largo de esta tesis. Pero que en ningún momento se ha tratado de innovar o modificar un sistema criptográfico, al contrario, todos los algoritmos utilizados han sido publicados y ratificados por expertos en su campo, siendo el uso aquí dado el expuesto por ellos.

3.2. *Stream Ciphers*

Se define un *stream cipher* como un algoritmo que permite cifrar bit a bit un texto en plano, *plaintext*, y obtener así el texto cifrado, *ciphertext*. En esta tarea de cifrado bit a bit, el *stream cipher* genera una cadena de bits denominada *key stream*, el *plaintext* usa esta cadena de bits para obtener el *ciphertext*. La manera de generar el *key stream* es lo que diferencia entre si a los distintos *stream ciphers*.

Para una correcta definición podemos definir los *stream ciphers* como un subconjunto dentro de los cifrados simétricos. Estos cifrados encriptan bit a bit, utilizando para ello los bits provenientes del *key stream*. Suponiendo que tenemos un *plaintext* tal que $X = x_0, x_1, \ldots, x_n$, obtenemos un *ciphertext* $Y = y_0, y_1, \ldots, y_n$ a partir de un *key stream* $S = s_0, s_1, \ldots, s_n$. Podemos definir las operaciones de encriptación y desencriptación como [PP10]:

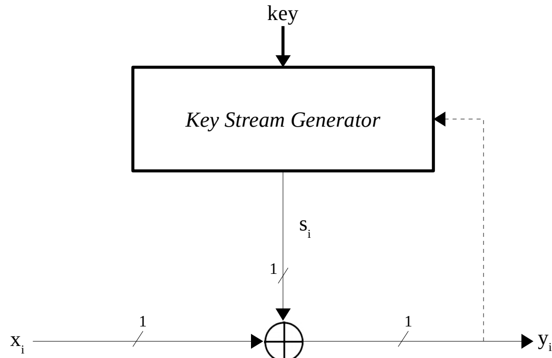


FIGURA 3.1: diagrama de un *stream cipher* sicrono/asíncrono

$$\textbf{Encriptación: } y_i = ENC_{s_i}(x_i) \equiv x_i \oplus s_i \mod 2 \tag{3.1}$$
$$\textbf{Desencriptación: } x_i = DEC_{s_i}(y_i) \equiv y_i \oplus s_i \mod 2 \tag{3.2}$$
$$\text{para} \quad x_i, y_i, s_i \in \{0,1\} \tag{3.3}$$
$$i, n \in \mathbb{N} \quad , \quad 0 \leq i \leq n \tag{3.4}$$

Dentro de los *stream ciphers* se pueden encontrar dos categorías. Si el *key stream* solo depende de la clave se denominan asíncronos. Si, por el contrario, el *key stream* depende tanto de la clave como del *ciphertext* se denominan síncronos. En la figura 3.1 podemos observar un diagrama de los *stream cipher*, la línea punteada hace referencia a los *stream ciphers* síncronos.

La seguridad de los *stream ciphers* depende en exclusividad de la generación de bits del *key stream*. Estos bits generados no son la clave de cifrado, básicamente es la computación del *stream cipher*. Por lo que la labor del *stream cipher* es conseguir que la secuencia de bits generada parezca lo mas aleatorio posible para un atacante.

3.3. *Block Ciphers*

La tarea de convertir un *plaintext* a *ciphertext* no es exclusiva de los *stream ciphers*. Existen alternativas, quizás la mas común entre distintos protocolos de seguridad sean los *block ciphers*.

Los denominados *block ciphers* son un subconjunto dentro de los cifrados simétricos. Los *block ciphers*, como su nombre indica, trabajan con bloques de datos. Usando la misma terminología, los bloques previos a la operación de encriptación se denominaran *plaintext*, al aplicar el algoritmo de *block cipher* generará un bloque de datos del mismo tamaño que denominamos *ciphertext*.

El proceso inverso sería la desencriptación. Estas transformaciones están determinadas por la elección del algoritmo y de la clave k utilizada [KR11]. Las funciones las notaremos a partir de ahora como:

$$x = ENC_k(y) \tag{3.5}$$

$$y = DEC_k(x) \tag{3.6}$$

Un *block cipher* está constituido por dos parámetros. El primero de ellos es el tamaño del bloque de datos b, y el segundo la longitud de la clave k en bits, *len(k)*.

Por tanto, dada una clave k, un *block cipher* obtendrá para el conjunto constituido por cada una de las 2^b entradas de bloques, una salida distinta para cada una de ellas. Por lo que podemos afirmar que un *block cipher* es una función biyectiva. Es decir, definiendo el conjunto de entradas como X y el conjunto de salidas como Y, podemos definir:

$$|X| = |Y| \tag{3.7}$$

$$\forall y \in Y : \exists! x \in X \Leftrightarrow f(x) = y \tag{3.8}$$

Con esto podemos establecer que b nos determina el tamaño del conjunto Y siendo este 2^b. Y *len(k)* indica las posibles permutaciones que podemos obtener del conjunto Y. Es por ello, que un *block cipher* con *len(k)* tiene un total de $2^{len(k)}$ posibles claves, generando cada una de ellas una permutación de tamaño 2^b. Sin embargo existen un total de $(2^b)!$ permutaciones diferentes para un tamaño de bloque de b-bits, lo que es aproximadamente $2^{(b-1)2^b}$. Por lo que un *block cipher* sólo genera un pequeña fracción de esas combinaciones, pero he de ahí que la tarea del diseñador sea ocultar este hecho [KR11].

La mayoría de los *block ciphers* que se han desarrollado están basados en el trabajo de Claude Shannon, en particular en su artículo titulado *Communication Theory of Secrecy System* de 1949 [Sha49]. En este artículo se introducían las ideas de *confusion* y *diffusion* para el diseño de cifrados.

Previamente a explicar estos conceptos debemos definir el termino *redundancy*. Un texto posee ciertas propiedades estadísticas inherentes, por ejemplo, un texto en inglés puede analizarse mediante la frecuencia de monogramas, bigramas o trigramas, ya que poseemos tablas con la frecuencia de aparición de estos en la lengua inglesa. Por lo tanto, un criptoanálisis de un texto cifrado por medio de cifrados de sustitución es susceptible a «ataques de diccionario» que se aprovechan de estas propiedades estadísticas del *plaintext*. Esto sería *redundancy*. Para Shannon, un buen cifrado debería enmascarar lo máximo posible la *redundancy* del *plaintext*.

La idea de *confusion* es hacer que la relación estadística existente entre la clave y el *ciphertext* sea lo más confusa posible. Por su parte, la idea de *diffusion* es hacer que un cambio en el *plaintext* genere un gran contraste en el *ciphertext* con el propósito de ocultar la *redundancia* del *plaintext*. Para estos conceptos

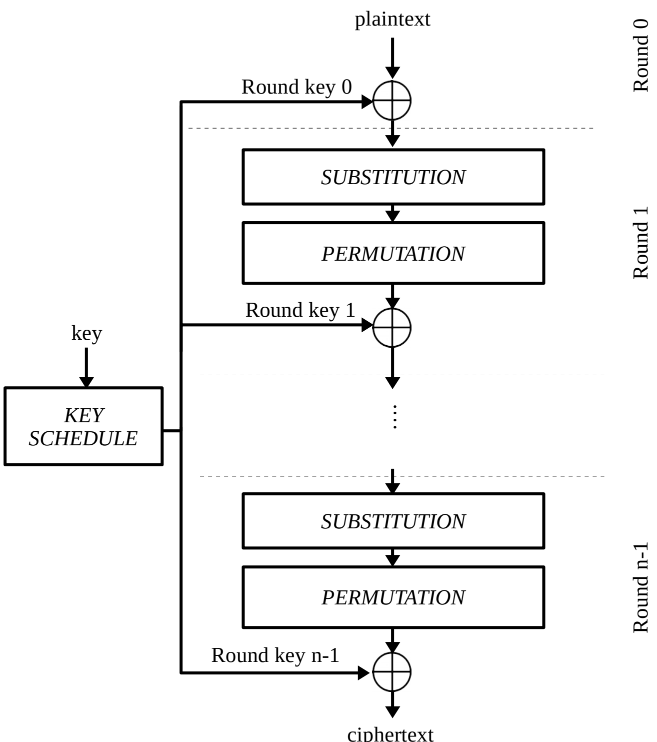


FIGURA 3.2: *SP-network*

se utilizan las operaciones de sustitución y permutación, para la *confusion* y *diffusion* respectivamente.

La operación de sustitución suele diseñarse con *lookup tables* o *substitution box*, referidas como *S-box*. Estas *S-boxes* están implementadas en memoria y deben operar a altas frecuencias para poder obtener el valor en el mínimo posible de ciclos. La permutación suele emplearse a nivel de bits, pudiendo obtener así una reordenación de cada bit.

La concatenación de operaciones de sustitución y permutación han dado lugar a una clase de *block ciphers* denominados *SP-networks*. Estos cifrados repiten en rondas la aplicación de unas determinadas operaciones de substitución y permutación, junto con la clave o una derivada de ésta. Es una buena práctica el reutilizar tanto de la clave original como sea posible. Para ello se utiliza el componente denominado *key schedule*, el cual genera una clave distinta denominada *round key* para cada ronda de substitución-permutación. En la figura 3.2 se pude apreciar un esquema de una *SP-network*.

3.3.1. Modos de Operación

Hay cinco modos de operación que todo *block cipher* puede adoptar. Cada modo provee de diferentes grados de seguridad y rendimiento. Estos modos son ECB, CBC, CFB, OFB y CTR. De todos ellos se procederá a explicar solamente aquellos implementados para la realización de la tesis, estos son

el ECB y el CTR. La elección de estos dos modos se debe a que el ECB es el modo básico, tal y como se explicara posteriormente, y el CTR dentro de los modos de operación que actúan como un *stream cipher* es el que mejor se adapta al cifrado de bloques no consecutivos.

En ambos modos consideraremos que el *plaintext* tendrá n bloques a los que denominaremos $m_1, m_2, m_3, \ldots, m_n$.

ECB

Este modo de operación es el básico, esto es porque en este modo se comporta tal y como se ha definido anteriormente, es decir, en el cual cada bloque se encripta de forma independiente. Se puede definir como [KR11]:

$$\left.\begin{aligned} c_i &= ENC_k(m_i) \\ m_i &= DEC_k(c_i) \end{aligned}\right\} \quad \text{para} \quad 1 \leq i \leq n$$

Esta condiciones hacen que el modo ECB pueda ser fácilmente paralelizable, consiguiendo así un alto rendimiento. La independencia entre bloques también conlleva a que si algún bloque se pierde, el resto se pueden desencriptar. Sin embargo, esto implica que la encriptación sea determinista. Una vez seleccionada la clave, un mismo bloque m_i siempre generará la misma salida c_i[PP10].

CTR

En este modo de operación el *block cipher* se comporta como un *stream cipher*. El modo CTR es utilizado para proveer s bits del *keystream* en cada iteración, siendo $s = b$. Por ello, se define como [KR11]:

$$\left.\begin{aligned} c_i &= m_i \oplus MSB_s(ENC_k(x_i)) \\ x_{i+1} &= INCREMENTO(x_i) \end{aligned}\right\} \quad \text{para} \quad 1 \leq i \leq n.$$

donde x_1 comienza con un valor aleatorio IV, y su comportamiento es fácilmente replicable con un contador. Como se aprecia de las ecuaciones cada bloque es independiente entre sí, pudiendo encriptar el que deseemos simplemente seleccionando el valor del contador con base IV. Un esquema de este modo se puede apreciar en la figura 3.3.

3.4. Funciones *hash*

Un aspecto importante en la transmisión de seguridad, es el conocer si el mensaje recibido a sido modificado, es decir saber del estado de integridad del mensaje. Es por ello que para esta tarea se utilizan las *funciones hash*.

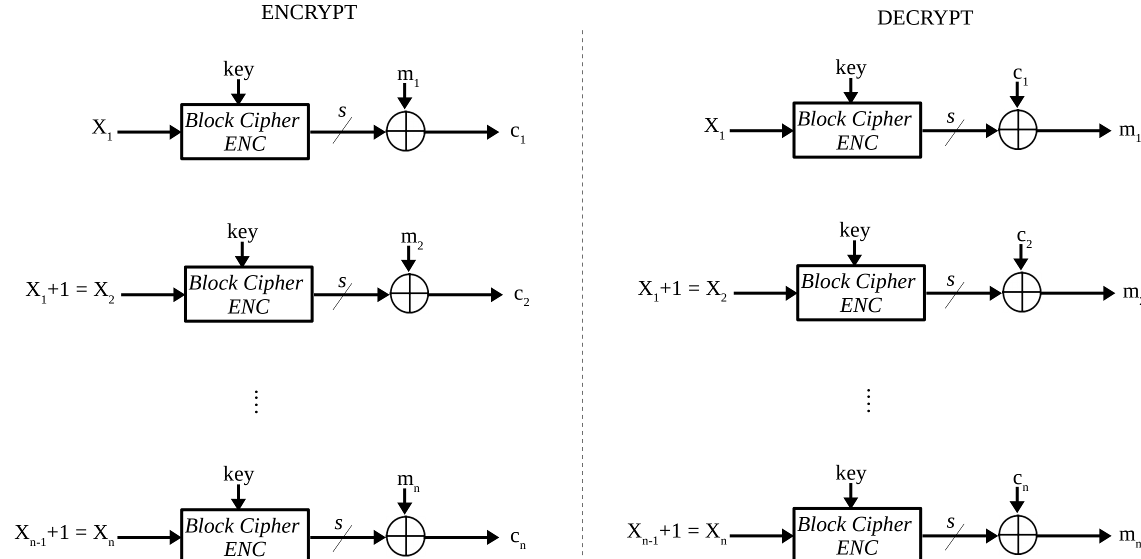


FIGURA 3.3: esquema del *CTR mode*

Una función *hash* es una función sobreyectiva, la cual toma como entrada un dato de tamaño arbitrario y devuelve un dato de longitud fija. Su propósito es generar un valor que represente al total de los datos de entrada, obteniendo así un resumen del mensaje o *digest*.

Una función *hash* debe poseer una serie de propiedades intrínsecas:

- debe poder procesar cualquier tamaño de entrada.
- debe producir un resultado de tamaño fijo.
- debe ser computacionalmente eficiente.

Además, para ser utilizada con fines criptográficos y con ello proveer al mensaje de integridad, debe cumplir una serie de requisitos, los cuales serán tratados en los subsiguientes apartados.

3.4.1. Clasificación de funciones *hash*

Las funciones *hash* se usan en un gran ámbito de aplicaciones, no todas ellas relacionadas con la criptografía. En estos casos se asume que estas funciones son *universal hash functions* [MF21]. Estas funciones se centran mas en la eficiencia que en la seguridad, es por ello que en esta tesis solo se definirá los denominados *criptography hash functions* o en lo que resta de documento *hash functions*. Estos últimos algoritmos deben cumplir una serie de requisitos para asegurar su validez criptográfica, estos requisitos/propiedades son:

- *Preimage Resistance*.
- *Second Preimage Resistance*.

- *Collision Resistance.*

3.4.2. *Preimage Resistance*

Las funciones *hash* deben ser *one-way*. Es decir, que teniendo el valor de salida de la función, $z = h(x)$. Sea computacionalmente impracticable obtener el valor de entrada x.

3.4.3. *Second Preimage Resistance*

Partiendo de que se ha calculado el *hash* del valor de entrada $x_1 = h(x_1) = z_1$. La propiedad de *Second Preimage Resistance*, también denominada *Weak Collision Resistance*, dicta que: debe ser computacionalmente impracticable obtener un valor de entrada x_2 tal que $x_2 = h(x_2) = h(x_1)$.

En teoría, al ser las funciones *hash*, funciones sobreyectivas es posible que exista x_2. Sin embargo, en general, un diseño *hash* tiene una resistencia ante este tipo de ataques de 2^n, donde n es el tamaño del dato de salida. En otras palabras, este tipo de ataques suelen ser por una búsqueda exhaustiva, por lo que no suponen un riesgo elevado escogiendo un valor apropiado para n.

3.4.4. *Collision Resistance*

Se denomina *Collision Resistance* a la propiedad de una función *hash*, tal que sea computacionalmente impracticable que, pudiendo elegir dos datos de entradas cualesquiera x_1 y x_2 siendo $x_1 \neq x_2$, se obtenga el mismo valor de salida para ambos, $h(x_1) = h(x_2)$.

Por tanto la cuestión que se plantea es cuantos mensajes hacen falta procesar para tener un porcentaje razonable de encontrar una colisión pudiendo seleccionar ambos datos de entrada.

Consideramos que la función *hash* genera valores de n bits. Seleccionamos t mensajes distintos.

$$\begin{aligned}
\Pr(\text{no colisión}) &= \frac{2^n - 1}{2^n} \times \frac{2^n - 2}{2^n} \times \cdots \times \frac{2^n - (t-1)}{2^n} \\
\Pr(\text{no colisión}) &= \prod_{i=1}^{t-1} \frac{2^n - i}{2^n} \\
\Pr(\text{no colisión}) &= \prod_{i=1}^{t-1} \left(1 - \frac{i}{2^n}\right)
\end{aligned}$$

Utilizando las series de *Taylor* podemos utilizar la siguiente aproximación:

$$
\begin{aligned}
e^{-x} \approx& 1 - x, \quad \text{sí y sólo sí } x << 1 \\
\Pr(\text{no colisión}) \approx& \prod_{i=1}^{t-1} e^{-\frac{i}{2^n}}; \quad \text{ya que } \frac{i}{2^n} << 1 \\
\Pr(\text{no colisión}) \approx& e^{-\frac{1+2+3+\cdots+t-1}{2^n}}
\end{aligned}
$$

La serie aritmética que aparece en el numerador del exponente se puede expresar como:

$$
1 + 2 + 3 + \cdots + t - 1 = \frac{t(t-1)}{2}
$$

Por lo que obtenemos el siguiente valor para la probabilidad de que no ocurra una colisión en t intentos.

$$
\Pr(\text{no colisión}) \approx e^{-\frac{t(t-1)}{2^{n+1}}}
$$

Una vez tenemos la expresión para la probabilidad de que no ocurra una colisión, calculamos la probabilidad de que ocurra al menos una.

$$
\begin{aligned}
\Pr(\text{colisión}) =& 1 - \Pr(\text{no colisión}) \\
\Pr(\text{colisión}) \approx& 1 - e^{-\frac{t(t-1)}{2^{n+1}}} \\
1 - \Pr(\text{colisión}) \approx& e^{-\frac{t(t-1)}{2^{n+1}}} \\
\ln(1 - \Pr(\text{colisión})) \approx& -\frac{t(t-1)}{2^{n+1}} \\
-\ln(1 - \Pr(\text{colisión})) \times 2^{n+1} \approx& t(t-1) \\
ln(\frac{1}{x}) =& \ln(1) - ln(x) = 0 - \ln(x) = -\ln(x) \\
t(t-1) \approx& 2^{n+1} \times \ln\left(\frac{1}{1 - \Pr(\text{colisión})}\right) \\
t^2 \approx& t(t-1); \quad \text{sí y sólo sí } t >> 1 \\
t \approx& \sqrt{2^{n+1} \times \ln\left(\frac{1}{1 - \Pr(\text{colisión})}\right)} \\
t \approx& 2^{\frac{n+1}{2}} \sqrt{\ln\left(\frac{1}{1 - \Pr(\text{colisión})}\right)} t \approx 2^{\frac{n}{2}}
\end{aligned}
$$

Como se observa podemos aproximar que el número de intentos para encontrar una colisión es $2^{\frac{n}{2}}$.

3.5. *MAC*

No solo es importante saber en la transmisión de un mensaje si este ha sido modificado (integridad), sino también saber quien es el emisor de dicho mensaje, por tanto hay que autenticar el mensaje (autenticidad), esta tarea fundamenta en la criptografía moderna corre a cargo de las funciones *MAC*.

Las funciones *MAC* al igual que las funciones *hash*, previamente descritas, generan un valor de tamaño fijo independientemente del tamaño del valor de entrada. La principal diferencia respecto a las funciones *hash* en que el valor generado es encriptado utilizando una clave k. Por tanto la siguiente notación sera utilizada:

$$y = MAC_k(x)$$

La forma más habitual de utilizar las funciones MAC es la descrita a continuación. Suponiendo una comunicación entre dos partes, que denominaremos A y B, donde ambas partes tienen en su poder la clave k y esta no ha sido comprometida. B envía un mensaje x hacia A, para ello, genera $m = MAC_k(x)$ y crea el mensaje $M = x, m$ donde concatena el resultado de la función *MAC* al mensaje. Este mensaje es recibido por A, el cual, utilizando la clave k calcula $m' = MAC_k(x)$ y comprueba que $m = m'$. En el caso de que sea correcto, A puede afirmar que el mensaje x no ha sido modificado, el mensaje posee la propiedad de integridad, y además que el valor m sólo ha podido generarse a partir del mensaje x y de una parte que posea la clave k, por tanto, el mensaje posee autenticación ya que hemos supuesto que la clave k no esta comprometida. Sin embargo, debido a que ambas partes deben tener en su control la misma clave k para la comunicación, k es una clave simétrica. No podemos asegurar en un mensaje si m ha sido generado por A o por B, por tanto el mensaje no tiene la propiedad de no-repudio [PP10].

Al realizar implementaciones de la funciones *MAC*, se pueden apreciar dos grupos bien diferenciados. El primero de ellos son las funciones *MAC* basadas en *block ciphers*, el segundo son aquellas que se basan en funciones *hash*. En esta sección describiremos con más detalle este último grupo, pues será la implementación escogida para los trabajos recogidos en esta tesis.

3.5.1. *HMAC*

Cuando la estructura principal de una función *MAC* esta basada es una función *hash*, se denomina *HMAC*. La idea básica de este diseño es utilizar la función *hash* para entremezclar el mensaje con la clave. Las construcciones más básicas son:

- *secret prefix MAC*: en esta configuración se concatena primero la clave junto con el mensaje. Teniendo pues, la clave como sufijo de la entrada

de la función *hash*.

$$m = MAC_k(x) = h(k||x) \tag{3.9}$$

- *secret suffix MAC*: en contrapartida, esta construcción, siendo prácticamente similar a la anterior, sólo varía en que la clave es usada como sufijo de la entrada de la función *hash*.

$$m = MAC_k(x) = h(x||k) \tag{3.10}$$

Sin embargo, ambas construcciones presentan serias carencias criptográficas que pasaremos a describir a continuación.

Para el caso de *secret prefix MAC*, el problema se basa en cómo se genera el *hash* para un mensaje X. Este mensaje será subdividido en bloques del tamaño de entrada de la función *hash* escogida, $X = x_1, x_2, \ldots, x_n$. Por lo que el HMAC será a su vez iterativo.

$$\begin{aligned} i &= 1 \\ HMAC(X) &= m_1 = h(k||x_1) \\ i &= 2 \\ HMAC(X) &= m_2 = h(m_1||x_2) \\ &\ldots \\ i &= n \\ HMAC(X) &= h(m_{n-1}||x_n) \end{aligned}$$

Como se puede observar, es posible para un atacante que tenga acceso a X y al resultado de la función *HMAC*, el poder añadir bloques al mensaje original y calcular su *HMAC* en función del último valor conocido, ya que en éste el valor de la clave k se vuelve irrelevante.

En el segundo supuesto, *secret suffix MAC*, observaremos que la clave k no nos aporta ningún tipo de mejora respecto a una función *hash*. Supongamos que somos capaces de encontrar una colisión tal que:

$$h(x) = h(x_o).x! = x_o$$

Entonces observamos que el *HMAC* de x y x_o será exactamente el mismo.

$$HMAC_k = h(x||k) = h(x_o||k)$$

Con las debilidades anteriormente descritas, el método por el que se ha decantado la implementación utilizada en esta tesis está basado en el trabajo de Mihir Bellare, Ran Canetti y Hugo Krawczyk presentado en 1996 [BCK96], el cual se explica en [PP10] de donde se ha recogido la información a continuación mostrada.

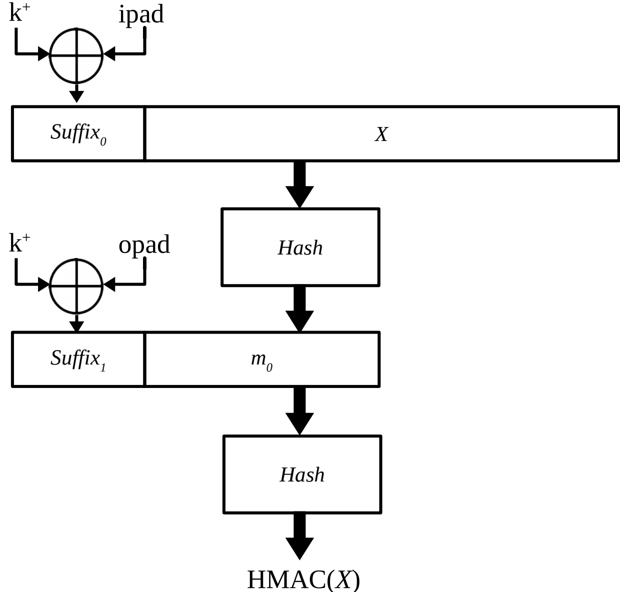


FIGURA 3.4: construccion *HMAC*

En esta construcción, la computación del *MAC* comienza añadiendo 0's a la izquierda de la clave k hasta tener b-bits. Esta clave extendida se le denominará k^+. El valor b viene dado por el tamaño del bloque de entrada de la función *hash*. Una vez obtenido k^+ se le realiza una operación XOR con el valor *ipad*.

$$ipad = 0x33, 0x33, \ldots, 0x33$$

El valor derivado de esta operación XOR es añadido como sufijo al mensaje X, como primer bloque del mensaje. Una vez realizado esto se procesa el *hash* de este nuevo mensaje:

$$m_0 = h[(k^+ \oplus ipad)||X]$$

Tras obtener este *hash*, se procede a calcular un segundo valor utilizando la función *hash*. Para ello, primero se vuelve a calcular un valor derivado de k^+ utilizando la operación XOR y un valor *opad*, cuyo patrón de bits es:

$$opad = 0x5C, 0x5C, \ldots, 0x5C$$

Una vez obtenido el resultado, se forma un dato de entrada para la función *hash* cuyo primer bloque es el resultado de la operación XOR y como segundo bloque se utiliza m_0. Obteniendo así el valor del *HMAC*:

$$HMAC_k = h[(k^+ \oplus opad)||m_0]$$

Un diagrama de esta construcción se muestra en la figura 3.4

3.6. *Key Derivation Function*

Una *Key Derivation Function* tiene como objetivo generar claves a partir de una serie de parámetros, entre ellos una clave maestra. Debido a que la clave maestra debe mantenerse en secreto, incluso para los poseedores de las claves generadas, una KDF debe tener la propiedad de ser una función de un sólo sentido, es decir con la salida y no se debe poder obtener los parámetros de entrada.

Las aplicaciones en las que las KDF son parte fundamental son diversas. Una de las principales es la de generar claves efímeras o *session keys*, estas claves tienen un tiempo de validez limitado y suelen emplearse para obtener una mayor seguridad en caso de que en una comunicación la clave quede expuesta, ya que esa clave sólo será válida para esa sesión. Sin embargo, el uso que se le va a dar en esta tesis es el siguiente: la KDF a partir de una clave maestra, la cual se usara para cifrar un mensaje m, genere claves de usuarios independientes entre ellas, con las cuales se posibilite un mecanismo para obtener m.

Para la definición de una KDF utilizaremos la mostrada en [YY05]. Una función KDF se denota por:

$$y = F(p, s, c)$$

donde cada parámetro representa:

- p: es la clave maestra, esta clave es el único parámetro que debe quedar en secreto para el resto de entidades.
- s: es el parámetro *salt*, el cual es un valor que es introducido en p para ocultar sus características en futuras transformaciones que se lleven a cabo.
- c: este parámetro representa el número de iteraciones que se llevarán a cabo dentro del KDF para poder generar la clave y.

La construcción de la función F usa como componente principal una función a la que se denominará H, este nombre viene dado a que por norma general el tipo de funciones están basadas en funciones *hash*.

Capítulo 4

Resumen global de los resultados

En este capítulo se presenta un resumen de los artículos que componen esta tesis. Tal y como se describió en la introducción, esta tesis está diferenciada en tres partes. La primera de ellas se centra en el desarrollo de un Core bootloader hardware para dispositivos IoT sobre FPGA, esta primera parte se sustenta en dos trabajos: una conferencia donde se presenta la aplicación [CQ18] y un articulo [GM+20] donde una segunda versión del bootloader es utilizada como base para el desarrollo de una modificación sobre el procesador OpenRISC [Ope]. La segunda y tercera parte nacen como solución a los problemas que se encontraron durante la realización de la primera parte y constituyen el eje central sobre el que se sustenta esta tesis por compendio. La segunda parte de la tesis presenta una metodología integral para la verificación, por simulación y On-Chip, de Cores sobre FPGA, todo este procedimiento esta respaldado por el articulo *An Integrated Digital System Design Framework With On-Chip Functional Verification and Performance Evaluation* [CQ+21a] presentado en el capitulo 5. Para finalizar, la tercera y ultima parte de esta tesis presenta un nuevo Core, E-LUKS, que permite la confidencialidad, integridad y autenticidad de los datos de usuario para dispositivos IoT, al igual que en la anterior parte este trabajo se sustenta en el articulo *Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security* [CQ+21b] incluido también en el capitulo 5.

4.1. Bootloader Hardware

Esta primera parte de la tesis se corresponde con los trabajos sobre el diseño del bootloader hardware sin la necesidad de recurrir a software. La aproximación inicial fue la de dotar a una FPGA de la capacidad de arrancar un sistema Linux de manera remota. Para ello se utilizó la interfaz de comunicación Wi-Fi. La idea principal consistía en tener un servidor, el cual contiene una imagen del sistema operativo y que enviaría dividida en segmentos del tamaño máximo permitido por el receptor. En el ámbito del receptor se optó por usar como puente, entre el servidor y la FPGA, el chip ESP8266 [Esp]. Este chip habilita de una manera relativamente sencilla la comunicación inalámbrica con el servidor y habilita una comunicación SPI con la FPGA, ejerciendo esta última como *slave*. Los principales desarrollos de este

trabajo sin embargo, se localizan en los Cores incluidos en la FPGA. Por un lado, se ha adaptado el modulo DDR2 utilizando para ello la herramienta MIG ofrecida por Xilinx [Mig] junto con un puente entre esta y el bus del sistema, este Core se basó en [Wb2], por ultimo el Core que más desarrollo atrajo fue el Core encargado del arranque, *Bootloader Module*. Este Core controlaba la señal de reset del procesador y ejercía de mediador entre los mensajes recibidos por el ESP8266, los cuales procesaba y enviaba los datos válidos a la memoria DDR. Una vez lograba almacenar la totalidad de la imagen, tras varios envíos, devolvía las señales de control de la memoria al bus del sistema y desactivaba el reset del procesador, el cual buscaba en la memoria el programa a arrancar. Los resultados arrojados por esta implementación mostraban que la cantidad de recursos utilizados por el *Bootloader Module* eran mínimos (3.84 % Slices), además de poder ser utilizado por cualquier otro procesador, consiguiendo por ello flexibilidad. Por contrapartida, la dependencia de utilizar un microprocesador externo para la comunicación, así como el tiempo llevado a cabo para la transferencia de la imagen completa, llevó a tener que refinar el diseño.

Este refinamiento se contempla en el artículo *Address-encoded byte order*. En este trabajo, la parte que compete al *Bootloader Module* es el de servir con el propósito de preparar el sistema, para llevar a cabo una modificación en el procesador y testarlo sobre un sistema Linux. Este nuevo bootloader utiliza una tarjeta microSD para almacenar la imagen del sistema operativo eliminando con ello las dos complicaciones: el tiempo de transferencia y la necesidad de un microprocesador externo. Esta implementación añadía un Core basado en [Cla+17] para actuar como *master* en la comunicación con la tarjeta microSD, así como para ejecutar comandos sobre ella. Las modificaciones de este Core se basaron en habilitar tarjetas clase 10, añadir comandos de lectura multibloque de la tarjeta, habilitar comandos de escritura y por último, reescribir el código en SystemVerilog en vez de VHDL, dividiéndolo de tal forma que fuera reutilizable mediante herramientas como fusesoc [Fus].

Estos trabajos fueron los desencadenantes de los siguientes artículos a tratar, los cuales constituyen la mayor parte de esta tesis. Tras estos, en el capítulo 6 se procederá a explicar la conexión.

4.2. Metodología de verificación de IPCores

Centrándonos en este artículo, tenemos como foco de interés una metodología la cual permite realizar diseños basados en IPCore en unos tiempos y con una fiabilidad acordes a los requisitos actuales de la industria.

4.2.1. Definición Conceptual de la metodología

Esta metodología consta de tres fases asociadas a tres niveles del diseño diferentes. La primera de dichas fases está asociada con el software de más alto

nivel, en ella se realiza un esbozo o esquema de la funcionalidad que tendrá el IPCore a diseñar. Para ello, se realiza en código de alto nivel esta funcionalidad, en nuestro caso en particular, se optó por Python. Esto permite tener una visión más concreta del diseño y familiarizarnos con él, además de poder utilizarlo como referencia para generar los patrones de test que más adelante se requerirán para llevar a cabo la verificación. La segunda fase está vinculada con el código RTL del diseño y su posterior simulación, el desarrollo del código RTL se ve influenciado por la fase anterior, la cual nos servirá como guía. Una vez tenemos una primera versión del código se procede a una simulación *white-box testing* utilizando para ello CocoTB, una herramienta para realizar simulación RTL con Python, favoreciendo una mayor rapidez de cubrimiento para realizar una verificación funcional del código, en caso de que la simulación sea correcta se procederá a la última fase, en caso contrario se modificará el código RTL y se continuará con otra simulación, constituyendo así un ciclo iterativo hasta su correcto funcionamiento. Por ultimo, la tercera fase está ligada a una verificación hardware *black-box testing*, es decir, comprobar el correcto funcionamiento del IPCore desarrollado en la fase 2 sobre la FPGA. Para esta labor se ha implementado un IPCore que realizará esta tarea, obteniendo los patrones de test de una memoria flash externa, una microSD, y comparando estos con los resultados generados por el IPCore. Este Core ha sido denominado como Autotest Core y será explicado con más detalle posteriormente. Como cabe esperar en caso de que la tercera fase sea infructuosa, la metodología insta a volver a la fase 2 para modificar el IPCore. Como ya se comentó anteriormente, los patrones de test utilizados en las dos últimas fases de la metodología son idénticos y ambos generados por el código software de alto nivel implementado en la primera fase. La figura 4.1 muestra una representación de las fases que componen esta metodología.

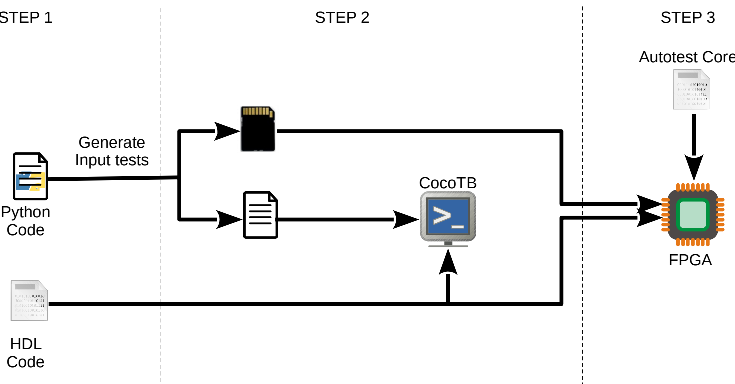


FIGURA 4.1: Representación de la metodología propuesta

4.2.2. Modulo Autotest Core

Como se puede observar, la novedad incorporada a esta metodología es el Core encargado de efectuar la verificación On-Chip, el Autotest Core. Para dicho cometido, este componente ha sido diseñado con el objetivo en mente

de poder abarcar el máximo número de IPCores y poder realizar la verificación de forma autónoma. Los patrones de test son almacenados en una tarjeta microSD, por lo que permite flexibilidad tanto en la naturaleza de estos como en su cantidad. Todos estos test contienen un valor esperado que el Autotest Core recoge y compara con la salida del IPCore bajo test (CUT), ya estimulado con los valores de entrada del patrón. Una vez obtiene una resolución del CUT, vuelve a almacenar este valor junto con métricas de rendimiento, en la tarjeta microSD junto con el patrón de test seleccionado. Esto hace posible un posterior análisis más detallado sin tener que incrementar la complejidad del Autotest Core. Un esquemático del Autotest Core es presentado en la figura 4.2.

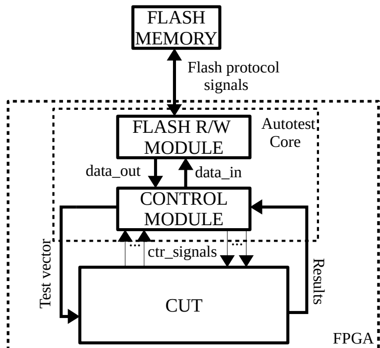


FIGURA 4.2: Esquemático del Autotest Core

Para una mayor adaptabilidad hacia los CUTs, el Autotest Core puede implementarse en dos modos diferentes según la naturaleza del CUT. El primero de estos modos sería el denominado *Single Block Mode*, este funcionamiento tiene como CUT objetivos aquellos que tienen parámetros de entradas de tamaño fijo. En contrapartida, tenemos el *Multiple Block Mode*, utilizado para aquellos CUTs cuyas entradas puedan tener un tamaño arbitrario, tal y como podrían ser las funciones *hash*. Para ambos casos han sido desarrolladas sendas plantillas, las cuales tras unos mínimos cambios se pueden adaptar al CUT según su naturaleza.

4.2.3. Resultados

Para comprobar la eficacia de esta metodología, especialmente la última de sus fases, se ha llevado a cabo con dos Cores que divergen en su naturaleza. Para testear el *Single Block Mode* se ha utilizado el *block cipher* PRESENT [Bog+07]. El cifrado PRESENT es un cifrado simétrico de bloques de 64-bits con una clave de 80-bits o 128-bits; para nuestro caso en particular se ha optado por la clave de 80-bits. En relación con el *Multiple Block Mode*, se ha seleccionado para testear la metodología la función *hash* SPONGENT [Bog+11]. Esta función *hash* genera una salida de N-bits, donde N puede ser 88, 128, 160, 224 o 256, en esta implementación se ha optado por seleccionar el valor de N igual a 88. En ambos casos se comenzó con un diseño de ambas

funciones en Python para guiar la siguiente etapa desarrollada en SystemVerilog. Una vez desarrollados los Cores se pasó a verificarlos funcionalmente mediante simulación RTL con la herramienta CocoTB. El último paso de verificación On-Chip se realizó además de con el Autotest Core, con la herramienta de Xilinx VIO, la cual ofrece una verificación *black-box testing* sobre la FPGA, siendo así una herramienta semejante a la aquí descrita. Por tanto, los resultados que se presentarán a continuación se centran en la última fase de la metodología así como en la comparación con otras herramientas de verificación. Cabe destacar que en el caso particular de PRESENT, ha sido posible establecer otra comparación con un diseño BIST centrado en este cifrado particular [Hai+19].

En ambos casos, los CUT han sido implementados en dos FPGAs distintas, la primera de ellas es la Xilinx Artix7 XC7A100T-1CSG324C [Xil10]. La segunda es la Xilinx Kintex7 XC7K325T FFG900-2 [Xil10]. Sin embargo, los resultados mostrados en este resumen sólo atañan a la FPGA Xilinx Artix7 XC7A100T-1CSG324C. Esto es debido al paralelismo mostrado en ambas FPGAs. Las métricas obtenidas en estos ejemplos, y por las cuales podremos extraer datos para su posterior análisis, son: el numero de recursos que toman las herramientas de verificación de la FPGA, la velocidad media para procesar un patrón y, por último, el tiempo total para procesar hasta 1000 patrones de test.

TABLA 4.1: Recursos tomados por diferentes herramientas de verificación funcional para el testado del *block cipher* PRESENT en la FPGA Xilinx Artix7 XC7A100T-1CSG324C.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Single Block Mode)	334	2.11	764	0.60	1023	1.61	0	0
Xilinx VIO Core	443	2.79	1655	1.31	842	1.33	0	0
BIST architecture proposed in [Hai+19]	33*	0.21	14*	0.01	163*	0.26	0*	0

* : indicates estimated values.

TABLA 4.2: Recursos tomados por diferentes herramientas de verificación funcional para el testado de la función *hash* SPONGENT-88 en la FPGA Xilinx Artix7 XC7A100T-1CSG324C.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Multiple block mode)	418	2.64	813	0.64	1362	2.15	0	0
Xilinx VIO Core	400	2.52	1469	1.16	833	1.31	0	0

Comenzando por la primera métrica asociada a los recursos de la FPGA, podemos observar la cantidad de recursos que toman las diversas herramientas de verificación On-Chip, excluyendo, por supuesto, la cantidad asociada al CUT. En la tabla 4.1 observamos los datos asociados al PRESENT y en la tabla 4.2 los datos del SPONGENT. Con estos resultados podemos determinar que el consumo de recursos debido a estas herramientas es inferior al 3 % del total, siendo las soluciones de Autotest Core y VIO muy similares y destacándose la solución BIST que, a costa de ser un Core extremadamente específico, consigue unos mejores resultados ocupando menos del 0.3 %.

Continuando con la velocidad media para procesar un patrón de test, estas gráficas incluyen CocoTB, simulando *black-box testing*, para mostrar una comparación entre las distintas herramientas. Las gráficas de las figuras 4.3 y 4.4 representan los tiempos para PRESENT y SPONGENT respectivamente. Ambas gráficas están en escala log 10 para poder así observar mejor la diferencia en ordenes de magnitud entre las distintas soluciones. Los resultados del PRESENT muestran que al igual que pasaba anteriormente la solución BIST al ser una solución especifica e inclusive un diseño del cifrado optimizado, consigue unos resultados 4 órdenes de magnitud inferior al Autotest Core. En cuanto a la diferencia entre el procesamiento de un patrón por el PRESENT Core, sin necesidad de nada mas, y del tiempo total con el Autotest Core, ésta se debe al tiempo requerido de entrada/salida por la tarjeta microSD en modo SPI. Aun así, Autotest Core está 2 órdenes de magnitud por debajo de las otras soluciones genéricas, siendo una de ellas On-Chip como es VIO. Respecto a los resultados de SPONGENT, cabe indicar que la salida es de 88 bits y las entradas que toman en los patrones de test son de 1024 bits. Una vez aclarado esto, se observa que los resultados son más favorables, siendo similares a los resultados de SPONGENT y de éste con Autotest Core, esto se debe a que el tiempo de procesamiento de un patrón es mayor que el tiempo de entrada/salida de la tarjeta microSD, siendo considerado en este caso como marginal. Por otra parte, Autotest mejora en 4 órdenes de magnitud a las soluciones de VIO y CocoTB, siendo por tanto este modo de funcionamiento, *Multiple Block Mode*, el mas óptimo respecto a otras soluciones.

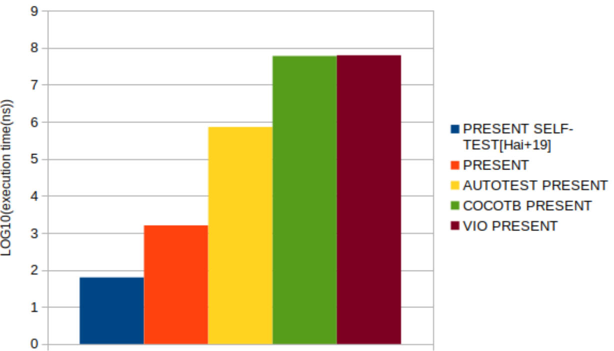


FIGURA 4.3: Tiempo medio de ejecución de un patrón para el Core PRESENT en escala log10.

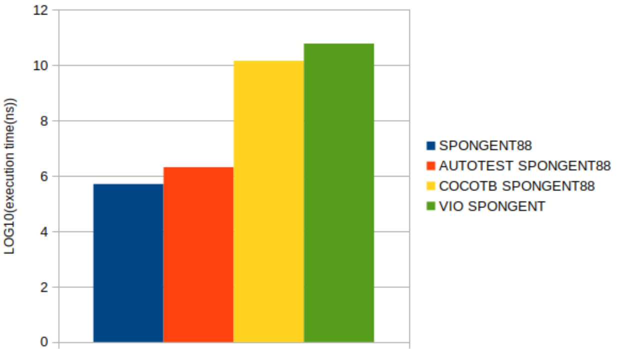


FIGURA 4.4: Tiempo medio de ejecución de un patrón para el Core SPONGENT-88 con entrada de 1024 bytes en escala log10.

La última métrica muestra el tiempo total en procesar distintas cantidades de patrones de test hasta un límite de 1000. La figura 4.5 muestra los tiempos para PRESENT y la figura 4.6 hace lo propio para los tiempos de SPONGENT. Al igual que en la métrica anterior, los parámetros de SPONGENT son: una salida de 88 bits y entrada de 1024 bits. Ambas gráficas demuestran que cuanto mayor sea el número de patrones a probar, más óptimo sera Autotest Core. Por lo tanto, podemos asumir que Autotest Core es una herramienta especialmente útil cuando se deben testar un gran número de patrones.

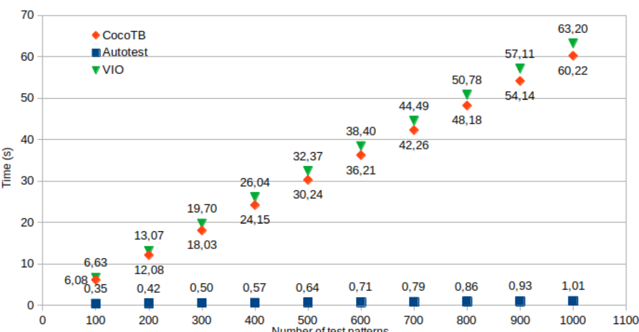


FIGURA 4.5: Tiempo de ejecución del total de patrones en el IPCore PRESENT.

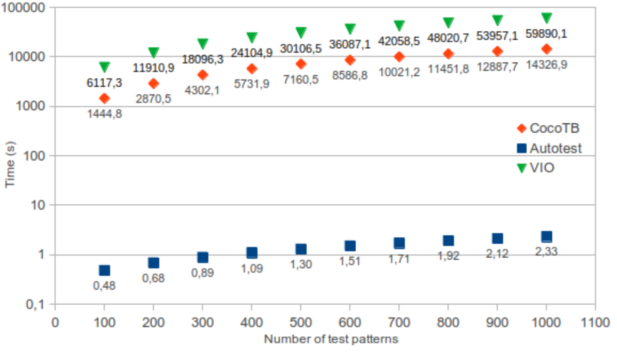


FIGURA 4.6: Tiempo de ejecución del total de patrones en el IPCore SPONGENT-88 con entrada de 1024 bytes.

4.3. *Embedded LUKS*

La tercera parte de esta tesis tiene como eje central la elaboración de un Core que habilita las propiedades de confidencialidad, integridad e autenticidad de los datos de usuario. Este Core ha sido desarrollado siguiendo la metodología anteriormente explicada.

4.3.1. Introducción

Embedded LUKS (E-LUKS) se desarrolló con el objetivo de procurar un protocolo similar al *Linux Unified Key System* (LUKS) [FB18] para dispositivos IoT. Por tanto, LUKS fue utilizado como base para el posterior desarrollo de E-LUKS. Ambos se basan en efectuar *Full Disk Encryption* (FDE) sobre los datos de usuario, usando para ello una clave maestra; dicha clave maestra es encriptada y almacenada para su posterior recuperación. La particularidad de LUKS es el hecho de que la clave maestra puede ser encriptada de diferentes formas relacionadas con distintas claves de usuario, estas claves permiten tener distintos usuarios accediendo a los datos sin tener acceso a las otras claves de usuario y sin conocer la clave maestra. E-LUKS ofrece confidencialidad, integridad y autenticidad, sin embargo, LUKS sólo ofrece confidencialidad. Si bien éste no es el primer acercamiento a ofrecer seguridad a dispositivos IoT, existen soluciones como: *Sancus* [Noo+13], *Soteria* [Göt+15], *Atlas* [Mae+19] o la presentada en [Wer+17]; con E-LUKS se obtienen unos resultados que le permiten ser una opción destacable entre éstas, esto se abordará cuando posteriormente se comparen los resultados referentes a los recursos utilizados en su implementación.

4.3.2. Diseño de E-LUKS

Centrándonos en el diseño de E-LUKS, lo primero que se propone modificar respecto a LUKS es limitar el tamaño de la estructura interna para poder adecuarse mejor a dispositivos IoT, y, primordialmente, modificar los algoritmos criptográficos y seleccionar aquellos adecuados para dispositivos IoT. Por tanto, comenzando con la estructura interna de E-LUKS, mostrada en la figura 4.7, observamos claramente que la partición de memoria se divide en una zona no encriptada y otra encriptada. La zona encriptada contiene los datos de usuario, mientras que la otra parte constituye el E-LUKS phdr, la cual contiene la cabecera de E-LUKS (header) y espacio para hasta 11 usuarios. La cabecera almacena distintos parámetros útiles para los diferentes algoritmos criptográficos, así como para identificar la partición. La tabla 4.3 muestra los campos de la cabecera. Los espacios para los usuarios (KS_x), contienen información sobre el usuario así como la clave encriptada y los parámetros requeridos, exceptuando la clave del usuario para su posterior recuperación. La tabla 4.4 muestra los campos de un KS_x.

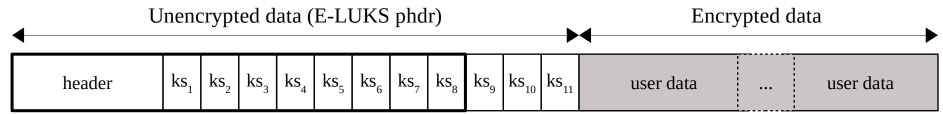


FIGURA 4.7: Estructura interna de E-LUKS.

TABLA 4.3: Campos de la cabecera E-LUKS.

offset(bytes)	nombre del campo	tamaño(bytes)	descripción
0	magic	6	identifica la partición como E-LUKS
6	mk-digest	11	resultado del KDF de la clave maestra (mk)
17	mk-digest-salt	8	parámetro salt para la clave maestra en KDF
25	mk-digest-iter	4	parámetro count para la clave maestra en KDF
29	mk-hmac	11	salida del HMAC de los datos encriptados de usuario
40	mk-IV	8	parámetro IV para el *block cipher*
48	user-data-blocks	4	bloques (512-bytes sector) de datos de usuario

TABLA 4.4: Campos de E-LUKS KS_x.

offset(bytes)	nombre del campo	tamaño(bytes)	descripción
0	activate	4	indica si KS_x esta activado
4	iterations	4	parámetro count para el KDF
8	salt	8	parámetro salt para el KDF
16	pwd-encrypted	16	*key material* (KM_x), clave maestra encriptada
40	IV	8	parámetro IV para el *block cipher* usada para generar pwd-encrypted

En lo concerniente a los algoritmos criptográficos, se han utilizado en E-LUKS como *block cipher* el algoritmo PRESENT, presentado anteriormente. Como función hash y eje central para la realización de la función HMAC, se ha optado por el algoritmo SPONGENT. Por último, en relación al KDF se ha usado el algoritmo presentado en [YY05]. Esto contrasta con la selección original de LUKS, siendo los algoritmos de E-LUKS eficientes para dispositivos IoT, aunque hay que tener en cuenta que la seguridad que aportan es inferior a los algoritmos utilizados en LUKS. Sin embargo, es más que suficiente para el entorno en el que está destinado a trabajar E-LUKS.

Una vez establecidas las principales cualidades de E-LUKS, pasaremos a describir su funcionalidad a través de las operaciones implementadas. La primera de estas operaciones es *Initialisation*, en la cual se generan los campos de E-LUKS phdr necesarios para la correcta funcionalidad de la partición. A continuación, se encuentra *Add New Password*. En esta operación se activa un usuario con su correspondiente KS_x, el cálculo para constituir *pwd-encrypted* se realiza de la siguiente manera (donde BC significa *Block Cipher* y MK representa la clave maestra):

$$
\begin{aligned}
&KeyUserDigest = KDF(KeyUser, salt, iterations)\\
&for(i = 0; i < \left\lceil \frac{MK_{Lenght}}{BC_{Lenght}} \right\rceil ; i = i + 1)\\
&\qquad X_i = Encrypt_{KeyUserDigest}(MK[(BC_{Lenght} - 1) * (i + 1) : i * BC_{Lenght}])\\
&\qquad pwd\text{-}encrypted = pwd\text{-}encrypted + (X_i \ll BC_{Lenght} * i)
\end{aligned}
$$

La siguiente operación es *Master Key Recovery*, la contrapartida de *Add New Password*, pues ésta última es requerida para que pueda ejecutarse. En ella, sólo si el usuario está activo puede recuperar con su clave la clave maestra encriptada. Después, estaría una operación incluida para E-LUKS denominada *HMAC Verification*. La finalidad de esta operación es verificar la autenticación e integridad de los datos de usuario a través del algoritmo criptográfico HMAC. Por último, se encuentra *Password Revocation*, con ésta operación desactivamos a un usuario, borrando con ello los datos asociados a KS_x.

Tras presentar E-LUKS se observa que está fuertemente basado en LUKS, por ello, la tabla 4.5 hace una referencia con todos los aspectos comunes de ambas soluciones. Con esto se establece una comparación necesaria para seleccionar adecuadamente la solución idónea para la situación requerida.

Para cerrar el diseño de E-LUKS, la figura 4.8 muestra el esquemático del Core, cada uno de los componentes ha sido testado con la metodología presentada al principio de este capítulo, no es por tanto de extrañar que los resultados de la metodología mostraran el cifrado PRESENT y la función hash SPONGENT como ejemplos. Esto ha conducido a que el Core haya sido resuelto de una forma adecuada y en unos plazos comedidos.

TABLA 4.5: comparativa de LUKS con E-LUKS.

		LUKS	E-LUKS
Criptografía	block ciphers	aes,twofish, serpert,cast5,cast6	PRESENT
	block ciphers (modos)	ecb,cbc-plain, cbc-essiv:*hash*, xts-plain64	ctr
	funciones hash	sha1,sha256, sha512,ripemd160	SPONGENT-88
	KDF	PBKDF2	KDF[YY05]
	HMAC	-	basado en SPONGENT-88
Estructura	tamaño cabecera (bytes)	208	52
	KS_x tamaño (bytes)	48	48
	phdr tamaño (bytes)	596	512
Operaciones	Initialisation	Sí	Sí
	Add New Password	Sí	Sí
	Master Key Recovery	Sí	Sí
	HMAC verification	No	Sí
	Password Revocation	Sí	Sí

4.3.3. Resultados

Con el fin de obtener unas métricas para E-LUKS se ha propuesto el siguiente proceso de verificación y testado. En este sentido, se ha implementado el Core en una FPGA y se ha formateado una tarjeta microSD con una partición E-LUKS, en dicha partición se almacenará como datos de usuario un fichero encriptado de un determinado tamaño. A continuación de esta partición, se almacenará el fichero en plano. Una vez preparada la tarjeta, la FPGA realizará tres comprobaciones: primero leerá el fichero en plano y lo almacenará en memoria, tras esto, usando el Core de E-LUKS, leerá el fichero encriptado y lo comprobará con el texto en plano. Por último, repetirá esta lectura pero añadiendo las propiedades de integridad y autenticidad de los datos.

Con este procedimiento de testado, podemos extraer las siguientes métricas: el tiempo de ejecución para cada uno de los casos y la utilización de recursos por parte del Core E-LUKS. Con la primera de las métricas podemos obtener el sobrecoste en tiempo que requiere E-LUKS, la tabla 4.6 muestra estos datos con un porcentaje del exceso de tiempo respecto a la lectura del fichero

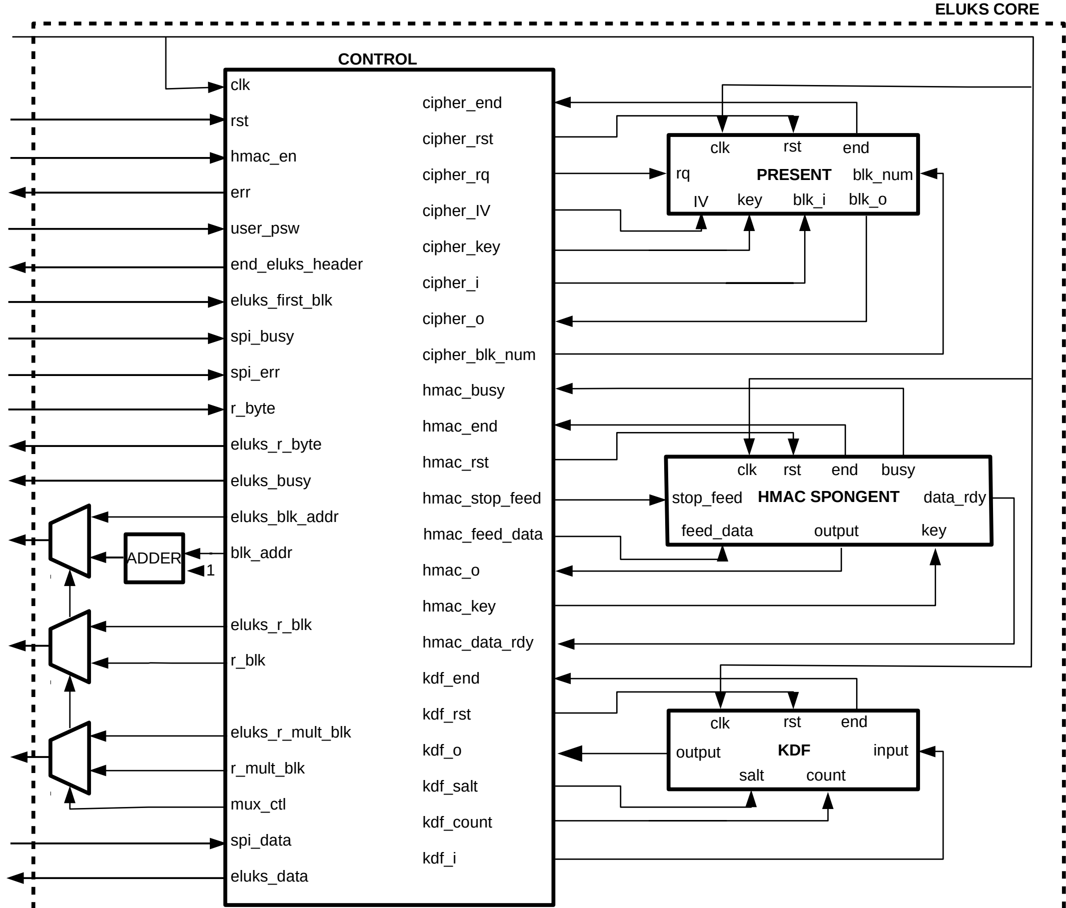


FIGURA 4.8: Esquemático del Core E-LUKS.

en plano. Se puede apreciar que ha medida que el fichero crece, este coste disminuye, y que cuanta más seguridad sea requerida (parámetro count y verificación HMAC), aumenta este coste. Respecto a la segunda métrica, se han realizado comparaciones por separado, primero se ha realizado una comparativa de recursos con implementaciones de LUKS sobre FPGA, la tabla 4.7 muestra dichos resultados. Se ha continuado con una comparación en recursos con alternativas que proveen seguridad de datos de usuario para dispositivos IoT, éstas se muestran en las tabla 4.8.

TABLA 4.6: Tiempos de ejecución para la XC7A100T FPGA.

tamaño del fichero	parámetro count	sin encriptar (tiempo en ms)	E-LUKS encriptado (tiempo en ms)	E-LUKS encriptado con HMAC (tiempo en ms)
4KB	32	1.81 (+0.0 %)	4.16 (+128.9 %)	6.8 (+275.7 %)
	64	1.81 (+0.0 %)	5.15 (+184.5 %)	7.40 (+308.8 %)
	128	1.81 (+0.0 %)	7.15 (+295.0 %)	9.39 (+418.8 %)
8KB	32	3.41 (+0.0 %)	6.75 (+97.9 %)	10.77 (+215.8 %)
	64	3.41 (+0.0 %)	7.75 (+127.3 %)	11.76 (+244.9 %)
	128	3.41 (+0.0 %)	9.74 (+185.6 %)	13.99 (+310.3 %)
16KB	32	6.58 (+0.0 %)	11.89 (+80.7 %)	19.90 (+202.4 %)
	64	6.58 (+0.0 %)	12.89 (+95.9 %)	21.13 (+221.1 %)
	128	6.58 (+0.0 %)	14.88 (+126.1 %)	23.50 (+257.1 %)

Respecto a los resultados de la comparativa con LUKS, queda en evidencia que E-LUKS es una solución mucho más óptima para dispositivos IoT, consumiendo una mínima parte de los recursos de la FPGA, menos del 5 %, en comparación con LUKS, que toma como mínimo un 25 % de los recursos de la FPGA.

TABLA 4.7: Comparación de recursos entre LUKS y E-LUKS para la XC7Z020 FPGA.

Core	FPGA	Slice LUTs (%)	Slice Registers (%)	BRAMs (%)
E-LUKS	XC7Z020	2672 (5.02)	2732 (2.57)	1 (0.74)
LUKS [Li+16]	XC7Z020	28068 (52.76)	29866 (28.07)	36 (25.53)
LUKS [Li+20]	XC7Z020	41656 (78.3)	66447 (62.45)	22 (15.47)

Sin embargo, en relación a los resultados de la comparativa con otras soluciones para IoT, observamos que, exceptuando la solución con autenticación de [Wer+17], todas ellas consumen una mínima parte de los recursos de la FPGA. Podemos apreciar también que sólo *Sancus* y *Soteria* tienen unos mejores resultados que E-LUKS, esto se debe a que estas funciones crecen a medida que aumenta el número de *SM* o *Software Modules*, básicamente cada *SM* constituye un programa dentro de la memoria del dispositivo. Sin embargo, al alcanzar los 10 *SM* los resultados para E-LUKS y estas soluciones se equiparan en termino de recursos. Por lo que podemos definir que la solución E-LUKS, que proporciona FDE, funciona con mayor eficiencia cuantos más ficheros distintos contenga la memoria. No obstante, es importante recalcar las propiedades de estas soluciones para establecer un baremo a la hora de seleccionar la solución adecuada, estas propiedades se recogen en la tabla 4.9.

TABLA 4.8: Comparativa de recursos para soluciones específicas de IoT. Los valores - no aparecen en los artículos originales.

Core	FPGA	Slice (%)	Slice LUTs (%)	Slice Registers (%)
E-LUKS	XC7Z020	946	2672 (5.02)	2732 (2.57)
[Wer+17] sin Autenticación	XC7Z020	-	4735 (8.9)	2447 (2.3)
[Wer+17] con Autenticación	XC7Z020	-	10214 (19.2)	4682 (4.4)
E-LUKS	XC6VLX240T	936 (2.51)	2724 (1.81)	2691 (0.89)
Sancus[Noo+13] with 1 *SM*	XC6VLX240T	-	751 (0.50)	1166 (0.39)
Soteria[Göt+15] with 1 *SM*	XC6VLX240T	-	792 (0.53)	1374 (0.46)
Atlas [Mae+19]	XC6VLX240T	2451 (6.50)	1025 (0.68)	5412 (1.80)

TABLA 4.9: Comparativa entre soluciones IoT.

	E-LUKS	[Wer+17]	*Sancus*	*Soteria*	*Atlas*
confidencialidad	Sí	No	Sí	Sí	Sí
integridad	Sí	Sí	Sí	Sí	No
autenticación	Sí	Sí	Sí	Sí	No
requiere modificar ISA	No	No	Sí	Sí	Sí

Capítulo 5

Publicaciones

En este capítulo se presentan las publicaciones que componen esta tesis. El orden en el que se presentan viene dado por su peso e importancia en este documento. Es por ello que las dos primeras publicaciones son aquellas que sustentan la gran parte de este trabajo: la metodología integral y el Core E-LUKS. Tras ellos viene el articulo publicado en revista que usa como base la segunda versión del Core bootloader. Por último, se presenta la conferencia donde se presento la primera versión del Core bootloader.

5.1. An Integrated Digital System Design Framework With On-Chip Functional Verification and Performance Evaluation

5.1.1. Breve resumen

En esta publicación se presenta un método para diseñar IPCores. Dicho método, aúna técnicas software de alto nivel con verificación On-Chip, para lograr una verificación integral. Se comienza con un diseño en alto nivel (Python) de la funcionalidad del IPCore, después se realiza la implementación en HDL. Este diseño en HDL sera iterativo hasta que sea verificado por simulación. Por último, se ha diseñado un IPCore específico para realizar la verificación On-Chip, el Autotest Core. Este Core permite la verificación del IPCore, además de medidas del tiempo de ejecución por patrón de test, de manera autónoma y ofreciendo libertad para aplicarle los patrones de test que el desarrollador considere adecuados. Por lo tanto esta metodología de desarrollo es integral y genérica, pudiendo así ser usada para multitud de IPCores.

5.1.2. Datos Revista

- Nombre Revista: IEEE Access
- Índice JCR(2020): 3.367(Q2)

- Fecha publicación: 1-12-2021
- DOI: https://doi.org/10.1109/ACCESS.2021.3132188

Multidisciplinary : Rapid Review : Open Access Journal

Received October 20, 2021, accepted November 28, 2021, date of publication December 1, 2021, date of current version December 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3132188

An Integrated Digital System Design Framework With On-Chip Functional Verification and Performance Evaluation

GERMAN CANO-QUIVEU, PAULINO RUIZ-DE-CLAVIJO-VAZQUEZ, MANUEL J. BELLIDO-DIAZ, DAVID GUERRERO-MARTOS, JULIAN VIEJO-CORTES, AND JORGE JUAN-CHICO

Department of Electronic Technology, University of Seville, 41012 Seville, Spain

Corresponding author: German Cano-Quiveu (germancq@dte.us.es)

This work was supported in part by the Ministerio de Industria y Competitividad of Spain under Project TIN2017-89951-P (BootTimeIoT), and in part by the European Regional Development Fund (ERDF). The work of German Cano-Quiveu was supported by VI Plan Propio de Investigacion y Transferencia de la Universidad de Sevilla (VI-PPITUS).

ABSTRACT This paper introduces a design and on-chip verification framework for IPCores in FPGA platforms. The methodology of the proposed framework is based on the development of a high level software model, an HDL description of the IPCore and the verification of the system under test by the Autotest Core, an on-chip verification core developed for this framework. The test pattern generation is done at the high level in software and used throughout the design and verification process. HDL simulation results can then be compared to on-chip results and get performance measurements from the Autotest Core. The Off-line testing is possible by using standard low-cost Flash storage (SD card). The proposed framework and methodology applied to PRESENT and SPONGENT cryptographic algorithms has shown over two orders of magnitude better performance than commercial tools like Xilinx's VIO and a hardware footprint of the verification cored below 3% of the available FPGA resources.

INDEX TERMS FPGA, framework, HDL, IoT, IPCore, on-chip, performance, verification.

I. INTRODUCTION

Nowadays embedded systems such as those found in smartphones, smart cities, medical devices, home automation or security systems are part of our daily life. Thus the Internet of Things (IoT) is one of the most active fields of research. It has been estimated than there will be more than 64 billion IoT devices connected in 2026 [1]. The increasing IoT demand has created new market opportunities, specifically in FPGA that is expected to reach a value of 7.5 billion $ by 2030 [2]. This has hardened time-to-market constraints, which is a problem despite the availability of external Intellectual Property Cores (IPCores) and high-level-software techniques and tools such as MyHDL [3], PyRTL [4], CocoTB [5] or VIvado HLS [6]. The impact is remarkably severe in terms of verification and validation, since these processes take a large portion of the development time. Because of this, bug implementations are not uncommon. In 2020 only 17% of the projects were in production with no known bugs [2]. Therefore, it is imperative to speed up the verification process. That is a challenging task, since circuit verification is a complex issue and a field of study of its own. A crucial task included in verification is functional verification. The objective of functional verification is to make sure the system carries out the task it was designed to. In the case of a digital system, during functional verification, it is fed with a set of input vectors and the corresponding outputs are compared with precomputed free-fault values. Obviously, for the most part this cannot be carried out comprehensively since the size of the input space grows exponentially with the input vectors length. A set of pseudo-random input test patterns of manageable size is then chosen so that, if the system produces the corresponding outputs, then the probability of a fault in the design is low. The device or algorithm generating these patterns is called a Test Pattern Generator (TPG), and the system fed by them is the Core Under Test (CUT). If the verification fails, inspecting the internal signals of the system is then convenient in order to find the source of the flaw. If the procedure or device employed to carry out the verification makes it possible to inspect such internal signals, then it is said to provide white-box-testing [7]. In contrast, only the external signals are monitored in black-box-testing.

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

One of the reasons why verification takes so much time and resources is that it must be carried out at every stage of the development cycle. Automated procedures can apply the same test patterns at every stage in order to minimize this time. If the system has been described using a Hardware Description Language (HDL), this can be carried out at the earliest stages through simulation. However, simulation is very time consuming, making it infeasible when the size of set of test patterns is high. Also, some flaws can only be detected once a physical instance of the CUT is available. Verifying a physical instance can be faster and more reliable, but it requires additional devices. For example, dedicated apparatus called Automated Test Equipment (ATE) can be used to test a physical instance. ATE can be very powerful, but also expensive and bulky. Because of that, in field verification with ATE can be difficult or impossible. Alternatively, the CUT and the verification device can be in the same board or even the same Integrated Circuit (IC). Several Verification IPCores (VIP) are available for this. Obviously, On-Chip verification devices can be expedited by using Reconfigurable Hardware (RH) such as FPGA. An example of On-Chip verification devices are Internal Logic Analysers (ILA). ILAs are less expensive than external logic analysers and do not require an out-chip to monitor signals. However, they consume a remarkable share of the IC resources. Another example is the Built-In-Self-Test (BIST) [8], that is a mechanism that permits a device to feed itself with a fixed set of input patterns and compare the corresponding outputs or a hash of the outputs with a precomputed stored value. Usually, BIST provides poor control on the test patterns. It is suitable when it is necessary to check the system's reliability along its lifetime once it has been deployed, but not in previous stages.

In this paper, the authors introduce a new framework to design, test and measure the performance of an arbitrary digital system by using automated TPG an On-Chip open source verification device called Autotest Core. This core provides high throughput during hardware verification, performance measurements and immediate test/fail results. The framework provides functional verification at every stage of the development cycle, making it possible to reuse the set of test patterns. Additionally, It is a general purpose framework that can be applied to any implementation technology. The TPG can be chosen according to what is most in accordance with the CUT, and there are no restrictions in the size of the set of test patterns.

II. RELATED WORK

The main advantage of using FPGA devices for the design and implementation of digital systems is the possibility to carry out hardware verification during the design process itself, even if it is intended to be implemented in an Application Specific Integrated Circuit (ASIC). Because of this, FPGA plays an important role onto the hardware functional verification of critical parts or even the whole system. To perform this On-Chip functional verification, FPGAs vendors provide several tools. For example, Xilinx provides black-box-testing tools such as the Virtual Input/Output (VIO) core [9] as well as white-box-testing tools such as ChipScope [10]. The first one is a customizable IPCore that can both monitor and drive internal FPGA signals in real time, while the latter is an ILA. In [11] the methodology to carry out white-box-testing on hardware using Chipscope is described. It details several benefits, but also points out the limited number of samples that can be recorded in the internal block RAMs. Other authors have developed alternatives to the use of ILAs in order to solve their drawbacks. An example of this is the educational works presented in [12] and [13] used to implement telematic FPGA laboratories. In [12] the ILA is replaced by an integrated microprocessor that controls the operation of the CUT (Circuit Under Test), feeds it with input patterns, stores its output and internal signals and sends them outside of the FPGA. The work presented in [13] is more specific because it exclusively verifies the On-Board execution of a program loaded into an embedded microcontroller. A Finite State Machine (FSM) is used to control the operation. One of the main drawbacks of the ILAs is the reduced number of data that can be stored due to the size of the block RAMs of the FPGAs. Different alternatives have been proposed to solve this problem. In [14] the use of a microcontroller (picoblaze [15]) is proposed. The signals that are considered necessary to verify the operation of the CUT are captured and sent out in real-time. This system is suitable for the functional verification of IPs that operate at low speed and generate results in a permanent and constant way over a long period of time. In [16] a start-stop system is applied so that when the internal memories are full, the operation is stopped, the data is extracted, and the system is restarted again. This is done by controlling the system clock signal, stopping and reactivating it as the memories fill up and are downloaded. The work that we have been discussing so far focuses on the On-Chip functional verification process itself.

The previous contributions make it possible to carry out functional verification on a physical prototype. However, the verification methodology must be comprehensive, covering all phases of the design process. Also, the synergy between the verification process carried out at each stage is essential since it improves their quality and reduces the design time. This holistic approach is expedited by VIPs. These are IPCores specifically designed to check the functionality of specific protocols, interfaces or functionalities both at a discrete level and in combination with other IPCores. An example of this type of VIP is presented in [17]. In some holistic verification methodologies, the input patterns and obtained results are reused in later design stages. To this end, a high-level software programming language is used to get the first description of the system. This makes it possible to take advantage of the power of software programming language to carry out functional verification. In [18] a methodology of this type is proposed based on the high-level description tools of Xilinx (Vivado HLS) that automates functional verification

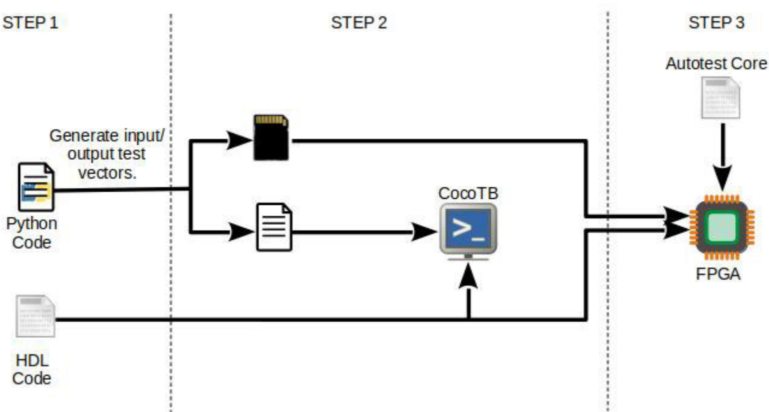


FIGURE 1. Design flow overview.

at Register Transfer Level (RTL), simulation and hardware levels.

In [19], the authors propose a co-validation design flow using the continuous integration methodology with tools such as Jenkins [20], GitHub [21] and Docker [22]. In this flow, a high-level software model of the system is used to verify each implementation. Another co-validation design flow is presented in [23]. In this paper, the authors introduce a python framework based on CocoTB called DUTILS to design SoCs. DUTILS is a python framework based on CocoTB which uses python to migrate high-level code to HDL. It uses a software model reference and from that creates by iterative steps the HDL model. Then, both are tested by the same test patterns, achieving an HDL module tested in a simulation-level. Reference [24] introduces another tool called LastLayer that makes it possible to create a C code simulation interface from an HDL description. In order to generate the interface, the HDL code must be adapted to a C library. According to the authors, this task is quite simple. Performing white-box-testing with the simulation interface is also straightforward, and the simulation interface is the same regardless of the implementation.

III. DESIGN FLOW OVERVIEW

From the analysis of previous work in the field of IPCore verification, three main questions arise that must be considered in any verification methodology:

1) **Test pattern generation.** As previously mentioned, most times an exhaustive test will not be possible, and a set of test patterns of manageable size must be carefully selected. Test pattern generation is a complex issue that is an active research field in its own, and is outside the scope of this paper.
2) **Functional verification**. The behaviour of the implementation fed by the selected test patterns is simulated at logical level. This has a high computational cost, and usually requires a remarkable share of the whole verification time.
3) **Hardware verification**. A physical instance of the implementation is fed by the selected test patterns.

Addressing these three main issues requires a comprehensive design, verification, and implementation procedure that begins with a high-level design and ends with a hardware verification of the implementation. The proposed approach is a fully integrated design and functional verification framework that consists of three main steps as shown in Fig. 1:

A. SOFTWARE LEVEL: DEVELOP A SOFTWARE MODEL OF THE SYSTEM

The design flow begins with the development of a high-level software model that describes the functionality of the system. The Python programming language is used for this task. The simplicity, clear structure and extensive availability of high-level programming libraries, makes it possible to write the software model in a fraction of the time required to design the hardware and verify it, taking advantage of the great facilities for the test that the Python language has. The software model also has the purpose to help the developer get a better understanding of the core functionality, making it possible to speed up the HDL design. The software model is then fed with a suitable set of test vectors, and the corresponding outputs are stored to be used as a reference in later steps. The proposed framework is flexible and does not impose a mechanism to get this set of test vectors. The developer can use test patterns or a TPG provided by a third party or write his own TPG. As previously mentioned, test pattern generation techniques are outside the scope of this paper.

B. RTL LEVEL: WRITE AND VERIFY THE HDL DESCRIPTION

First, the HDL description has to be developed. Then, the behaviour of the HDL description is simulated by using the CocoTB tool (RTL simulation). CocoTB is a co-verification Python tool that provides white-box testing. Specific python test code is written for critical parts of the HDL description, such as internal registers or counters at each clock cycle.

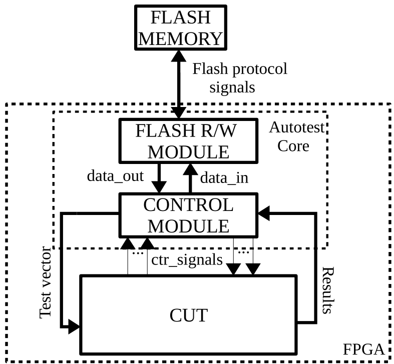


FIGURE 2. Proposed On-Chip functional verification system.

The overall functional verification of the whole HDL description is done by feeding CocoTB with a subset of the test patterns generated in step 1 and comparing its outputs with those generated by the software model.

C. HARDWARE LEVEL: VERIFICATION AND PERFORMANCE TESTING OF THE HARDWARE

In the last step, the system is implemented in a RH chip, alongside the so called Autotest Core, that will carry out black-box-testing. The Autotest Core is connected to the CUT inputs and outputs, as shown in Fig. 2. The test vectors and results previously generated by the software model are stored in a low-cost Flash memory device (microSD card). The Autotest Core then checks the functionality and performance of the CUT by feeding it with the test vectors, and comparing its outputs against the expected results. Every mismatch is reported and stored back in the microSD card alongside the performance measurements for further off-line analysis.

IV. AUTOTEST CORE

The proposed On-Chip functional verification system is shown in Fig. 2. In this Figure are three components; the flash memory which stores all the test records, the CUT which is the target to test on-board and finally the Autotest Core. The Autotest Core gets the test records from the flash memory by the Flash R/W Module using the SPI protocol. Once a test record is retrieved, it is processed by the Control Module to feed the CUT. To achieve it, the Control Module must be in charge of the control signals of the CUT, such as the reset signal, in order to get the CUT outputs which are compared with the expected ones. Lastly, the data collected from the CUT are send it from the Control Module to the Flash R/W module which writes it back into the flash memory.

Regards to the submodules of the Autotest Core, the Control Unit it is detailed below. However, the Flash R/W module is a modified version of the *minsdhost* module described in [25]; the differences are:

1) It is written in SystemVerilog instead of VHDL.
2) It supports class 10 microSD cards.
3) It implements the write command (CMD24 [26])
4) It implements the read multiple blocks command (CMD18 [26])

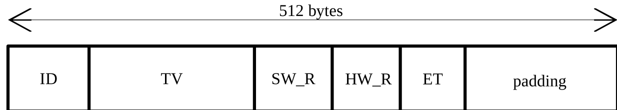


FIGURE 3. Record stored in the flash memory in the single block mode.

The Autotest Core description must be configured for each CUT. The Control Module has two operation modes suitable for different combinations of CUT and dataset. Both modes are different enough to provided an HDL implementation for each one. In order to agilize the design cycle template codes has been created and are available in the authors' github [27]. Therefore, when the designer chooses the dataset for the CUT then the appropriate template will be selected creating a synthesis of the full system.

A. SINGLE BLOCK MODE

This is the mode used to verify and measure the performance of any core whose input can be read in a single clock cycle, i.e. all the input bits are provided in parallel. The record corresponding to each test vector is stored within a single block of the flash memory using the format shown in Fig. 3. A block can only contain data for a single record, and test vectors to the same core are stored in a list of contiguous blocks.

Each record includes the following fields:

- `ID`: Every core to be tested is assigned a 32-bit identifier. The identifier of the core corresponding to the test vector is stored in this field. This makes it possible to store test vectors of different cores in the same flash memory.
- `TV`: This is the test vector to be applied to the CUT.
- `SW_R`: The expected output computed by the software model is stored in this field.
- `HW_R`: The output of the CUT is stored in this field.
- `ET`: The number of clock cycles employed by the CUT to generate the output is stored in this 64-bit field.
- `padding`: The size of this field is chosen so the size of the whole record is 512 bytes, i.e. the size of a block of the flash memory. Its content is not relevant since it is never read nor written.

The flow diagram for the single block mode is shown in Fig. 4. `initial_block`, `IPCUT_ID` and `max_timer` are parameters of the Autotest Core code template, while `current_b` and `errors` are local variables. `initial_block` is the number of the block containing the first test vector of the CUT, `IPCUT_ID` is the core identifier of the CUT and `max_timer` is an upper bound on the number of clock cycles required by the CUT to compute the output. Respecting the local variables, `current_b` encodes the number of the block containing the current input vector to be applied, while `errors` is the number of wrong outputs found. As previously mentioned, we used the seven segment displays available in our FPGA prototyping board to show `errors` in runtime.

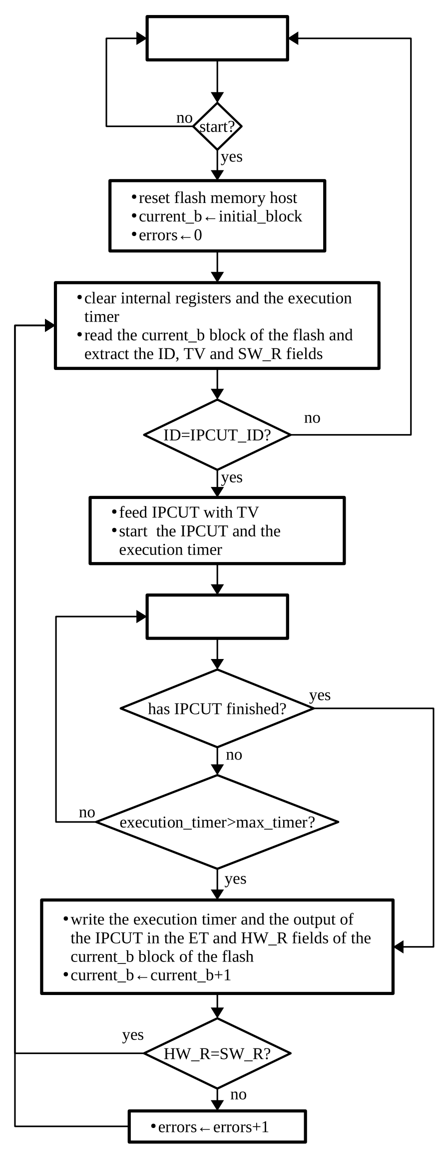


FIGURE 4. **Flow diagram for the single block mode.**

The behaviour of the Control Module in the single block mode is to wait until the start signal is activated. Then, it enters into a setup stage where the Flash R/W module is reset as well as all counters and registers. After that, the loop begins to check each test record. First, the test record fields are stored from the Flash memory into internal registers. Then, it makes a comparison between the ID field with a pre-stored signature, if both values are different then the Control Unit is finished, else the CUT is fed with the inputs from the test record and waits until an end signal is generated from the CUT. Due to the nature of the black-box testing, a timer has been introduced to be able to continue even if an internal state of the CUT is stalled. This timer also gives us the execution time that takes the CUT to process the data. Then, the output from the CUT is compared with the expected output, if both values are different then an internal counter for errors is incremented. Once the output and the execution time from the CUT are stored in the memory flash together with the test record, an internal register is updated to the upcoming block in the memory flash where the next test record can be found and the loop begins once again.

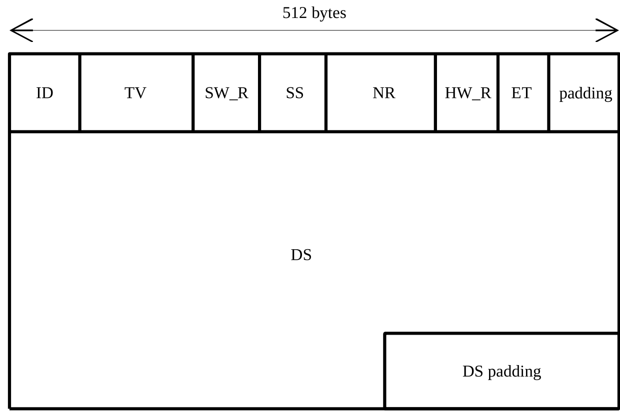


FIGURE 5. **Record stored in the flash memory in the multiple block mode.**

B. MULTIPLE BLOCK MODE

This is the mode to be used to verify and measure the performance of cores used to process a data stream of variable size such as checksum generators and cryptographic hash function implementations. The Autotest Core code template for this mode has an additional parameter labeled `word_size` that is the size in bytes of the words used to feed the CUT. As shown in Fig. 5, for each test pattern a record with the following fields is stored in the flash memory:

- `ID`: Every core to be tested is assigned a 32-bit identifier. The identifier of the core corresponding to the test vector is stored in this field. This makes it possible to store test patterns of different cores in the same flash memory.
- `TV`: If the input of the CUT has fixed-size fields, they are stored here. For example, it can be the key used by message authentication code generators or digital signature implementations.
- `SW_R`: The expected output computed by the software model is stored in this field.
- `SS`: The size in bytes of the data stream is stored in this 64-bit field.
- `NR`: The number of the first flash block used to store the record of the following test pattern, if any, is stored in this 32-bit field.
- `HW_R`: The output of the CUT is stored in this field.
- `ET`: The number of clock cycles employed by the CUT to generate the output is stored in this 64-bit field.
- `padding`: The size of this field is chosen so the size of the whole record (except the data stream) is 512 bytes, i.e. the size of a block of the flash memory. Its content is not relevant since it is never read nor written.
- `DS`: The data stream is stored in this field.
- `DS padding`: The size of this field is chosen so the size of the data stream is a multiple of 512 bytes, i.e. the size of a block of the flash memory. Its content is not relevant since it is never read nor written.

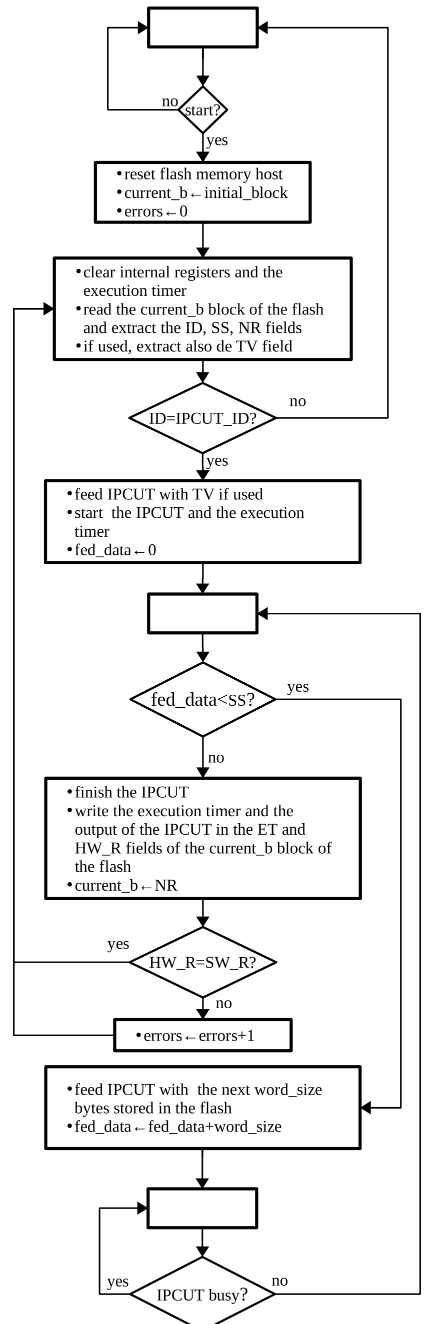


FIGURE 6. Flow diagram for the multiple block mode.

The flow diagram for this mode is quite similar to the previous one as shown in Fig. 6.

The behaviour of the Control Module in the multiple block mode is quite similar to the single block mode. Both make the same steps at the beginning until the way to feed the CUT. To feed the CUT in this mode, a chunk of data is read and then applied to the CUT which generates a new internal state. This process continues until all bytes stored have been applied. Then, the Control Module activates an end feed signal to the CUT which processes the data and when it finishes, it activates an end signal. It is important to clarify that a timer has been placed in the Control Unit as well as in the single mode. However, to get more accuracy in the result of execution time of the CUT, the timer is paused when the Flash R/W retrieves the data. Once the output is generated, it is stored together with the execution time of the CUT into the flash memory. If the generated output is different from the expected output, then the error counter is increased. After that, an internal register is updated with the value stored in the test record which indicates the upcoming block in the flash memory where the next test record is found. Lastly, the loop begins once again.

V. RESULTS

To validate the proposed approach and to estimate its performance, the following aspects have been evaluated: (i) Ease of integrating Autotest Core in the functional verification process of different systems; (ii) Resources used by the distinct operation modes; And (iii) Performance results (total execution time used to process a set of input test patterns in single and multiple block modes). Furthermore, these aspects have been compared with Xilinx VIO, an alternative method to perform black-box-testing on-board. In addition, one of the examples presented has been compared with a specific BIST solution detailed in [28]. However, this comparison can be only performed with the first example due to the specificity of the BIST solution.

A. EASE OF INTEGRATION

To demonstrate the ease of integrating Autotest Core in the functional verification process of any system, this approach has been used to design and verify a variety of IPCores available at the authors' GitHub repository. Most of these IPCores are hardware implementations of lightweight cryptographic algorithms:

- Block Ciphers: PRESENT and Twofish.
- Stream Cipher: Trivium.
- Hash Functions: Hirose PRESENT and SPONGENT.

These hardware implementations are perfectly adapted to the IoT perspective, where most devices are highly resource constrained and hardware implementations of traditional cryptographic algorithms cannot be afforded in terms of resources and power consumption. Therefore, once verified, these IPCores will be used to provide adequate security to different IoT devices that are being developed.

In order to organize them, a folder structure was used for each IPCore. The folders are:

- python_code: This folder contains the Python implementation of the IPCore functionality. In addition, it contains the TPG (Python script) which generates the test patterns.
- hdl_code: This folder contains the SystemVerilog implementation of the IPCore.
- cocotb_files: This folder contains a Python file which acts as a testbench file. This file is used for testing the SystemVerilog implementation with the test patterns generated by the TPG. A Makefile needed by CocoTB is also included in this folder.

- Hardware_verification_files: This folder contains two subfolders:
  - fusesoc_files: This subfolder contains a core file alongside the SystemVerilog top file. The core file collects all the files that FuseSoc [29] requires to generate the specific project files, depending on the selected Electronic Design Automation (EDA) tool: Vivado (Xilinx), Quartus (Intel), etc. As a result, this tool builds the bitstream file for the chosen hardware platform, which includes the IPCore as CUT and the Autotest Core as the functional verification core.
  - microSD_script_files: This subfolder contains all the necessary scripts to analyze the microSD card, once the Autotest Core has finished.

Moreover, this repository includes a user guide, which details the process of Autotest Core integration. Thus, it can allow designers to use this core to verify their own system.

The fact of providing all this information is twofold. Firstly, to demonstrate that Autotest Core is a generic approach that enables the functional verification of a wide range of systems quickly and easily, regardless of the technology used. And secondly, to facilitate that all the results presented in this paper can be contrasted.

Specifically, the obtained results in the functional verification process of two IPCores is shown in this paper: PRESENT block cipher and SPONGENT hash function.

The PRESENT IPCore is a SystemVerilog implementation of the PRESENT symmetric block cipher [30] with a block size of 64 bits and a key size of 80 bits. First, the HDL code of this block cipher was verified comparing the software results with the CocoTB simulation results. Later, the Autotest Core was adapted to the PRESENT cipher; and since inputs can be provided in parallel, the single block mode was used to verify this IPCore. The test vectors (`TV` record) have the following format:

- key: the 80-bit encryption/decryption key
- text: a 64-bit plaintext or ciphertext
- mode: a 1-bit control signal that selects the PRESENT Core mode to cipher (0-value) or decrypt (1-value)

The SPONGENT IPCore is a SystemVerilog implementation of the SPONGENT hash function [31]. This implementation may be fed with an arbitrary amount of input data. SPONGENT has an N-bit output where N can be 88, 128, 160, 224 or 256 bits. Therefore, a multiple block mode was used in order to test this IPCore. Input data is provided in the `DS` record; the `TV` record is empty.

Related to the VIO verification, the VIO IPCore can be easily configured and integrated using the Vivado Design Suite; and it is used to drive data into your design and read data from your design through the JTAG port. To automize the process of sending and reading data, an external software application is usually used. Unlike the Autotest Core, VIO is technology dependent and can only be implemented on Xilinx FPGAs.

```
import numpy as np
for k in range(0,N):
  # random TV fields
  key=np.random.randint(0,2**63-1,1,dtype=np.int64)
  text=np.random.randint(0,2**63-1,1,dtype=np.int64)
```

Listing 1. TPG simplified for PRESENT.

```
import random
for i in range(0,N):
  state = spongent_sw.initial_state()
  for k in range (0,stream_size):
    data_stream = random.randint(0,255))
    state=spongent_sw.feed_data(data_stream , state)
    microSD.write(data_stream)
```

Listing 2. TPG simplified for SPONGENT.

B. TPG SELECTION

As mentioned earlier, the TPG selection can be chosen by the developer. In our particular case, the goal is to test the proposed framework. Therefore, the TPG selection falls into the background. A TPG which generates a number of random test records has been used for simplicity. However, theoretically this particular TPG method has a high fault coverage assuming the single stuck-at fault model when applied to crypto-cores [32].

The following pieces of code are simplified versions of the python script used as TPG in these results, the full code can be accessed in the authors' github at the paths:

- block_ciphers\present_cipher\python_code\gen_testbench.py
- hash_functions\spongent_iter\python_code\gen_testbench.py

Listing 1 is the code for the PRESENT which has three TV fields. However, the mode field can be only 0, 1, therefore in each iteration of the TPG we create two records, one for each value of the mode field. Listing 2 is the code for the SPONGENT which only needs to generate the DS values. Each time data from the Stream is generated, it is used to feed the software model which updates the hash state. Once it is finished the random data is stored in the microSD.

C. RESOURCES

Regards to the hardware implementation on FPGA, both IPCores were implemented in a Nexys4DDR development board from Digilent [33], which included a Xilinx Artix7 XC7A100T-1CSG324C [34] FPGA chip. Besides this, in order to compare the amount of hardware resources used for our approach with those obtained in the PRESENT BIST work presented in [28], the PRESENT IPCore was also implemented in a Genesys2 development board which included a Xilinx Kintex7 XC7K325T FFG900-2 [34]. Similar results were obtained for both FPGAs. In this way, we assumed the same values is the Present BIST work for both boards in order to compare the results.

In table 1 the amount of resources used for three different alternatives in order to verify a PRESENT block cipher core (Autotest, VIO and the PRESENT BIST work) are shown.

TABLE 1. FPGA resources on Xilinx Artix7 XC7A100T-1CSG324C with the total percentage of the FPGA resources used by different functional verification cores in order to verify a PRESENT block cipher core.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Single block mode)	334	2.11	764	0.60	1023	1.61	0	0
Xilinx VIO core	443	2.79	1655	1.31	842	1.33	0	0
BIST architecture proposed in [28]	33*	0.21	14*	0.01	163*	0.26	0*	0

* : indicates estimated values.

TABLE 2. FPGA resources on Xilinx Kintex7 XC7K325T FFG900-2 with the total percentage of the FPGA resources used by different functional verification cores in order to verify a PRESENT block cipher core.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Single block mode)	332	0.65	764	0.19	1021	0.50	0	0
Xilinx VIO core	438	0.86	1655	0.41	842	0.41	0	0
BIST architecture proposed in [28]	33	0.06	14	0.003	163	0.08	0	0

TABLE 3. FPGA resources on Xilinx Artix7 XC7A100T-1CSG324C with the total percentage of the FPGA resources used by different functional verification cores in order to verify a SPONGENT88 hash function core.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Multiple block mode)	418	2.64	813	0.64	1362	2.15	0	0
Xilinx VIO core	400	2.52	1469	1.16	833	1.31	0	0

TABLE 4. FPGA resources on Xilinx Kintex7 XC7K325T FFG900-2 with the total percentage of the FPGA resources used by different functional verification cores in order to verify a SPONGENT88 hash function core.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Multiple block mode)	406	0.80	792	0.19	1339	0.66	0	0
Xilinx VIO core	411	0.81	1469	0.36	834	0.41	0	0

The results indicated that the BIST implementation is the option that uses the least hardware resources. This is because the PRESENT cipher core is reused as a TPG, therefore, being a specific solution. However, it can be seen that Autotest and VIO options are similar in this category. These approaches are generic solutions and the resource constraints are not critical since both cores use less than a 3% of the total slices. Similarly in table 2 the BIST implementation is the most efficient solution at the cost of being the more specific. Respect the other solutions although they are more expensive in resources, both are below the 1% of slices for the Xilinx Kintex7 XC7K325T FFG900-2 FPGA.

In tables 3 and 4 the amount of resources used by two different functional verification cores (Autotest and VIO) in order to verify a SPONGENT hash function core with an 88-bit output are shown. It can be seen that both cores have a low impact on the total FPGA resources, occupying less than 3% of the total slices for the Xilinx Artix7 XC7A100T-1CSG324C and less than 1% of the slices for the Xilinx Kintex7 XC7K325T FFG900-2. Therefore, in terms of hardware resources, they are generic solutions that allow designers to verify these types of systems.

Finally, as described in the previous sections, the Autotest Core read the input tests and expected results from the Flash memory device, so the whole functional verification process could be carried out within the FPGA. Regards to VIO core, an external script in TCL format was used to send the test vectors to the CUT through a JTAG port. This script also took care of receiving the output from CUT, comparing it with the expected result and storing all verification results in a file. Since the input tests were read one by one, the use of BRAMs was not necessary as observed in Tables 1,2 and 3. Therefore, hardware resources are highly optimized for both functional verification cores (Autotest and VIO) and operation modes (single and multiple block modes).

D. PERFORMANCE RESULTS

In this section, performance results will be presented. In the context of this paper, performance will be measured as the time dedicated to process a set of test patterns.

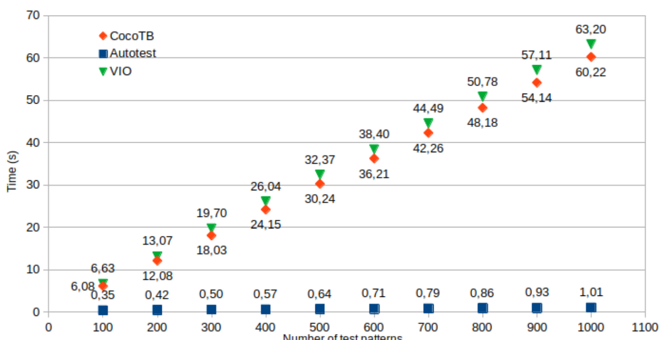


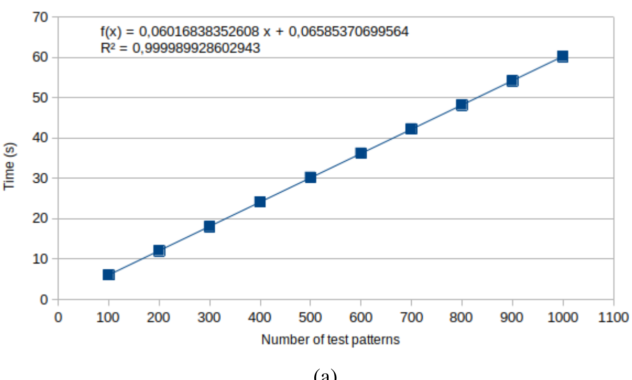
FIGURE 7. PRESENT IPCore. Execution times, partial up to 1000 test patterns.

As described in Section III, first, the behaviour of the HDL description was simulated using the CocoTB software tool. Next, was the On-Chip functional verification procedure. Therefore, three different alternatives were analyzed: CocoTB software tool, Autotest Core and Xilinx VIO core. All the software executions were executed in a computer with an AMD Ryzen 7 2700 Eight-Core Processor and 32 GB of RAM memory.

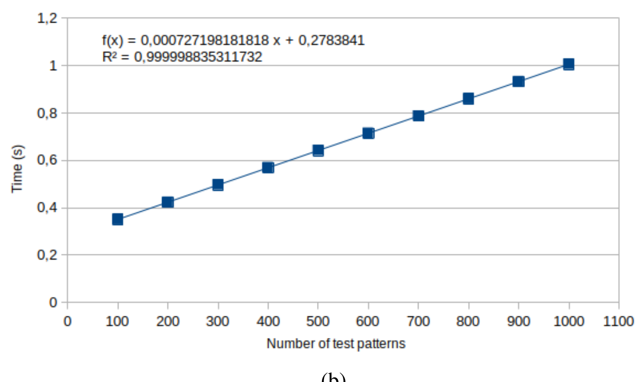
First, we can see the functional verification results of PRESENT block cipher are shown. Fig. 7 depicts execution times, partial up to 1000 test patterns (500 encrypts and 500 decrypts). This figure shows a similar performance for CocoTB software tool and VIO core. This is because the script used to communicate to the computer with the VIO core was running in software, so most of the time was spent reading test vectors from a file, exchanging data through the JTAG port and storing all functional verification results in another file. In this way, the potential of doing On-Chip functional verification using the VIO core was reduced in terms of communication with the computer. As seen in Fig. 7, this fact does not occur when Autotest Core is used, since all operations (reading of the test patterns from the SD card, pattern processing, etc.) are fully performed in hardware. This statement implies that Autotest Core can process a large number of test patterns per unit of time and perform a more thorough functional verification of systems, compared to the other alternatives.

From the data shown in Fig. 7, trend lines and coefficients of determination can be calculated for each alternative: CocoTB simulation tool, Autotest Core, and Xilinx VIO core (Fig. 8(a), 8(b), and 8(c), respectively). In these figures, the line slopes represent the average time per pattern, and the line constants represent the initialization times.

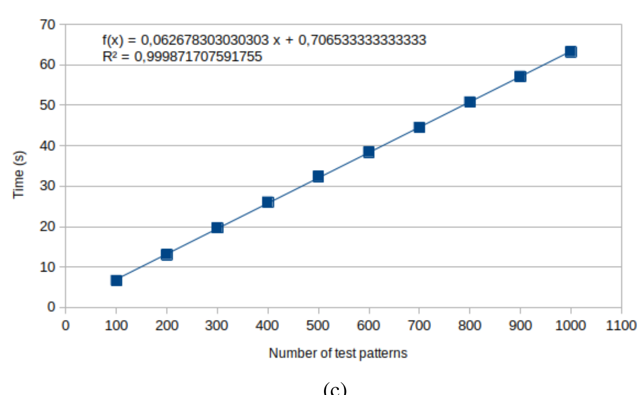
To get a better understanding of the results, Fig. 9 has been added. In this figure, it has been compared the average execution time (ns) taken to test one record. In addition, it has been included the time that the PRESENT cipher Core takes to encode or decode data in order to be able to compare both the time dedicated to the CUT, and the complete system. The values of the Y-axis are in logarithm base 10 to get a quick sight of the differences in orders of magnitude. The results show that the PRESENT BIST implementation presented in [28] has the better performance with an order of $10^{-8}s$ by far of the second best that is Autotest Core with 10^{-4}



(a)



(b)



(c)

FIGURE 8. PRESENT IPCore. Trend lines and coefficients of determination for each alternative: (a) CocoTB, (b) Autotest Core, and (c) Xilinx VIO core.

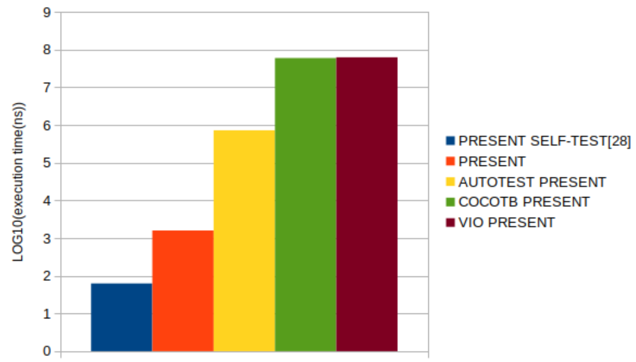


FIGURE 9. PRESENT solution average execution time for one record in log10 scale.

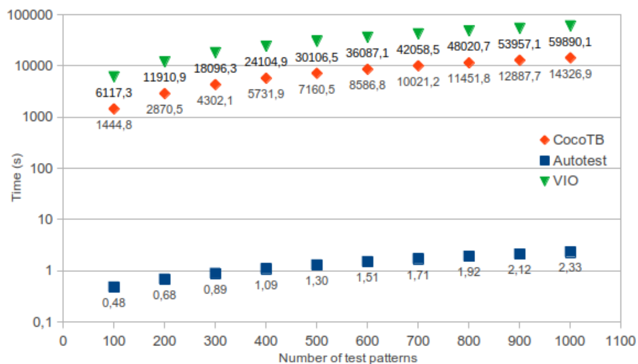


FIGURE 10. SPONGENT88 IPCore. Execution times, partial up to 1000 test patterns. The input data has a size of 1024 bytes.

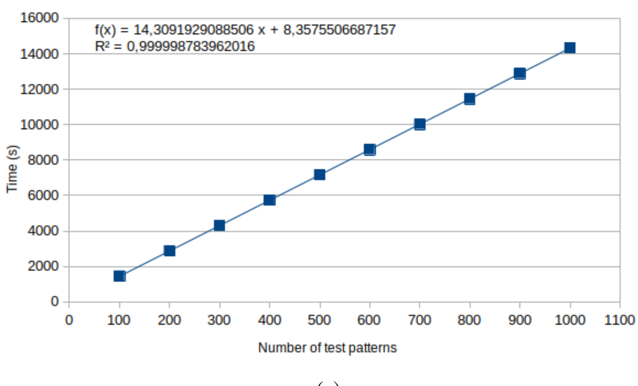
TABLE 5. PRESENT IPCore. Number of test patterns processed in one hour.

	Number of test patterns
CocoTB	59831
Autotest Core	4950125
Xilinx VIO core	57425

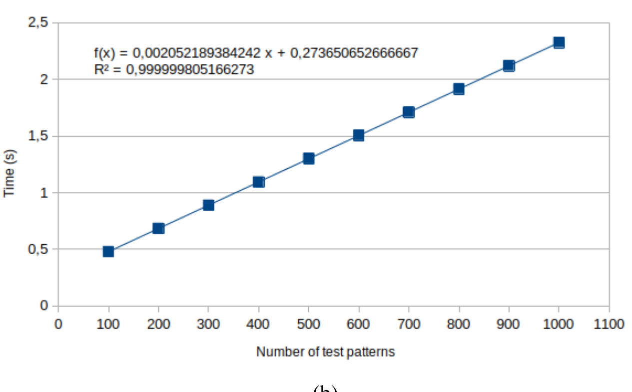
and the 10^{-2} of VIO. However, the clock signal used in the BIST solution is five times greater than the one used in the other cases, 500Mhz in opposition of 100Mhz. The conclusion with these results is that specific solutions are better in performance, but our framework is a better generic solution to test a broad set of Cores.

To demonstrate the power of Autotest Core, Table 5 shows the number of test patterns processed in one hour by each alternative. As seen in Table 5, the number of test patterns processed by Autotest Core exceeds the results obtained by CocoTB and VIO core by two orders of magnitude; speeding up the functional verification process. In addition, the type of functional verification that Autotest Core performs has two additional benefits. Firstly, that performing such a thorough hardware functional verification can allow designers to detect errors occurring after several hours of testing under certain established conditions. And secondly, it can check the maximum operating frequency of the system and measure the processing time of the system at that frequency. In this way, it has been verified this IPCore works properly at a maximum frequency of 400 MHz, getting a processing time of 390 ns.

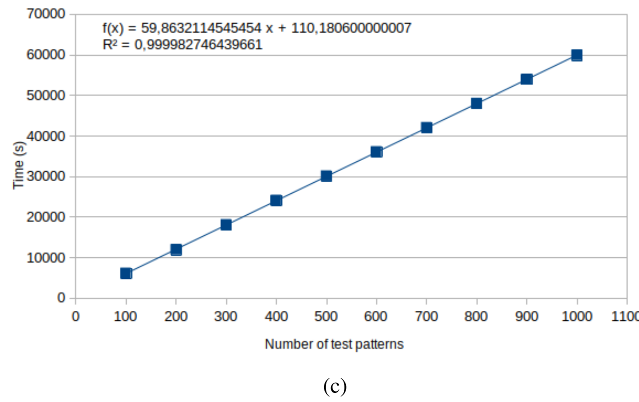
Secondly, the functional verification results of SPONGENT hash function will be shown. Fig. 10 depicts execution times, partial up to 1000 test patterns. The input data has a size of 1024 bytes. Trend lines and coefficients of determination for each alternative are shown in Fig. 11(a), 11(b), and 11(c). Regarding the VIO core, Fig. 10 shows that when the CUT must be fed with an arbitrary amount of input data (only 1024 bytes in this scenario), the performance of this core is very low; even CocoTB presents a better performance. This is because the VIO core is designed to replace or augment board-level I/O components such as status indicators and low-bandwidth controls (LEDs, buttons or DIP switches); but



(a)



(b)



(c)

FIGURE 11. SPONGENT88 IPCore. Trend lines and coefficients of determination for each alternative: (a) CocoTB, (b) Autotest Core, and (c) Xilinx VIO core.

it is not optimized to perform this type of hardware functional verification. Fig. 10 also shows Autotest Core can verify this type of IPcores in a reduced time. Besides, according to the Fig. 11(a), 11(b), and 11(c), the number of test patterns processed by Autotest Core exceeds the results obtained by CocoTB and VIO core by four orders of magnitude. This means that in scenarios where a larger number of test patterns or larger input data have to be processed, Autotest Core still offer a good performance.

In the same line with the previous example, Fig. 12 has been added. This figure shows the execution time (ns) to process one test record in the different solutions, alongside the time taken by the Core to generate the hash value.

TABLE 6. Power consumption.

Complete System	Dynamic Power(mW)	Static Power (mW)	Total Power (mW)
PRESENT Autotest Core	39	97	137
PRESENT VIO	15	84	99
PRESENT BIST [28]	63	83	146
SPONGENT88 Autotest Core	46	97	143
SPONGENT88 VIO	18	84	102

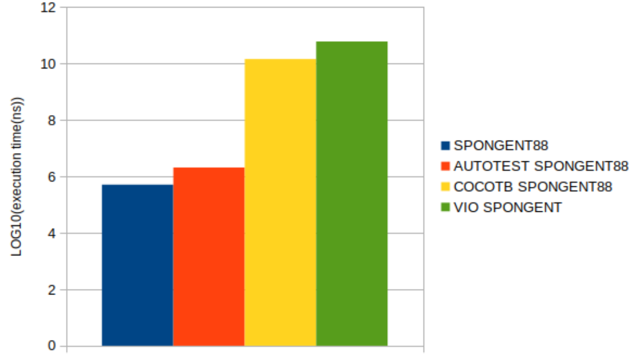


FIGURE 12. SPONGENT88 solution average execution time for one record in log10 scale.

The logarithm base 10 in the results is to compare the orders of magnitude. It is clear that Autotest Core has an advantage over VIO by four orders of magnitude, so the Autotest Core has an order of $10^{-3}s$ and VIO has an order of $10^{1}s$.

Finally, the performance of the SPONGENT IPCore has been determined by Autotest Core. In this way, this IPCore works properly at a maximum of 400 MHz, getting a processing time of 126,75 μs.

E. POWER CONSUMPTION

In this section, the power consumption will be presented. These results are an estimation performed by Vivado Power Analyzer tools.

With the values presented in table 6 we can establish that all the proposed solutions have the same order of consumption ($10^{-2}W$), being that one assumable in the IoT field.

VI. CONCLUSION

The verification framework proposed in this paper is adequate to design and test IPCores in a fast and reliable way by combining test pattern generation using a software model, HDL simulation, on-chip testing and performance measurements.

TPG generation, including expected results, is done at a high level in software and used throughout the whole verification process by the HDL model and the on-chip verification core (Autotest Core) greatly reducing verification time.

PRESENT and SPONGENT cryptographic cores and random pattern generation have been used to test the framework functionality successfully. It has been shown that the Autotest Core hardware footprint is below 3% of the total slices available in standard FPGA chips while the ability to read patterns and store results in low-cost standard Flash memory (SD card) provides a very flexible and cost-effective verification system.

Comparison to commercial alternatives like Xilinx's VIO shows that the Autotest Core performance is at least two orders of magnitude faster in addition to be able to do off-line testing without the support of an external system (computer), making it a great alternative to standard tools.

REFERENCES

[1] *IoT Infrastructure Technology & Connectivity Explained—Business Insider*. Accessed: Dec. 2020. [Online]. Available: https://www.businessinsider.com/iot-infrastructure-technology?IR=T

[2] H. D. Foster, "Quantifying FPGA verification effectiveness," *Verification Horizons*, vol. 16, no. 3, 2020.

[3] *MyHDL*. Accessed: Oct. 2020. [Online]. Available: http://www.myhdl.org/

[4] *PyRTL by UCSBarchlab*. Accessed: Oct. 2020. [Online]. Available: https://ucsbarchlab.github.io/PyRTL/

[5] *Introduction—COCOTB 1.1 Documentation*. Accessed: Aug. 2020. [Online]. Available: https://cocotb.readthedocs.io/en/latest/introduction.html

[6] Xilinx. (2018). *Vivado High-Level Synthesis*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[7] S. Nidhra, "Black box and white box testing techniques—A literature review," *Int. J. Embedded Syst. Appl.*, vol. 2, no. 2, pp. 29–50, Jun. 2012.

[8] C. E. Stroud, *A Designer's Guide to Built-in Self-Test* (Frontiers in Electronic Testing), 1st ed. New York, NY, USA: Springer, 2002.

[9] Xilinx. *Virtual Input/Output v3.0 LogiCORE IP Product Guide Vivado Design Suite*. [Online]. Available: http://www.xilinx.com

[10] *ChipScope Integrated Logic Analyzer (ILA)*. Accessed: Jul. 2020. [Online]. Available: https://www.xilinx.com/products/intellectual-property/chipscope_ila.html

[11] K. Arshak, E. Jafer, and C. Ibala, "Testing FPGA based digital system using Xilinx ChipScope logic analyzer," in *Proc. 29th Int. Spring Seminar Electron. Technol.*, May 2006, pp. 355–360.

[12] D. Garijo and R. Senhadji, "CcLAB: A tool for remote verification of FPGA-based circuits," *IEEE Latin Amer. Trans.*, vol. 14, no. 3, pp. 1115–1121, Mar. 2016.

[13] K. Saksida and A. Trost, "Remote laboratory for testing processor cores in FPGA device," in *Proc. 37th Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, May 2014, pp. 172–177.

[14] J. Viejo, J. I. Villar, J. Juan, A. Millan, E. Ostua, and J. Quiros, "Long-term on-chip verification of systems with logical events scattered in time," *Microprocessors Microsyst.*, vol. 36, no. 5, pp. 402–408, Jul. 2012, doi: 10.1016/j.micpro.2012.02.005.

[15] *PicoBlaze 8-bit Microcontroller*. Accessed: Jan. 2021. [Online]. Available: https://www.xilinx.com/products/intellectual-property/picoblaze.html#overview

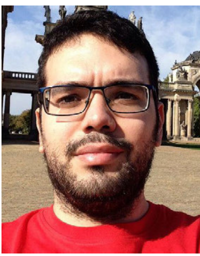
[16] H. U. H. Khan and D. Göhringer, "FPGA debugging by a device start and stop approach," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Nov. 2016, pp. 1–6.

[17] S. Harutyunyan, T. Kaplanyan, A. Kirakosyan, and H. Khachatryan, "Configurable verification IP for UART," in *Proc. IEEE 40th Int. Conf. Electron. Nanotechnol. (ELNANO)*, Apr. 2020, pp. 234–237.

[18] J. Caba, F. Rincón, J. Barba, J. A. De La Torre, J. Dondo, and J. C. López, "Towards test-driven development for FPGA-based modules across abstraction levels," *IEEE Access*, vol. 9, pp. 31581–31594, 2021.

[19] L. Beaulieu, O. Weppe, B. Le Ludec, and F. Lebeau, "Co-verification design flow for HDL languages: A complete development methodology," in *Proc. 24th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2017, pp. 530–533.

[20] *Jenkins*. Accessed: Nov. 2020. [Online]. Available: https://www.jenkins.io/
[21] *GitHub: Where the World Builds Software GitHub*. Accessed: Nov. 2020. [Online]. Available: https://github.com/
[22] *Empowering App Development for Developers | Docker*. Accessed: Nov. 2020. [Online]. Available: https://www.docker.com/
[23] M. Trapaglia, R. Cayssials, L. De Pasquale, and E. Ferro, "Flexible software to hardware migration methodology for FPGA design and verification," in *Proc. 10th Southern Conf. Program. Log. (SPL)*, Apr. 2019, pp. 39–44.
[24] L. Vega, J. Roesch, J. Mcmahan, and L. Ceze, "LastLayer: Toward hardware and software continuous integration," *IEEE Micro*, vol. 40, no. 4, pp. 103–111, Jul. 2020.
[25] P. Ruiz-de-Clavijo, E. Ostúa, M.-J. Bellido, J. Juan, J. Viejo, and D. Guerrero, "Minimalistic SDHC-SPI hardware reader module for boot loader applications," *Microelectron. J.*, vol. 67, pp. 32–37, Sep. 2017, doi: 10.1016/j.mejo.2017.07.007.
[26] SD Group. (2005). *SD Specifications Part 1: Physical Layer Specification. V1.10*. [Online]. Available: https://www.sdcard.org/downloads/pls/simplified_specs/part1_410.pdf
[27] *GitHub—Germancq/IPCores*. Accessed: Feb. 2021. [Online]. Available: https://github.com/germancq/IPCores
[28] Z. Haider, K. Javeed, M. Song, and X. Wang, "A low-cost self-test architecture integrated with PRESENT cipher core," *IEEE Access*, vol. 7, pp. 46045–46058, 2019.
[29] *GitHub—Olofk/Fusesoc: Package Manager and Build Abstraction Tool for FPGA/ASIC Development*. Accessed: Aug. 2020. [Online]. Available: https://github.com/olofk/fusesoc
[30] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Berlin, Germany: Springer, 2007, pp. 450–466.
[31] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede, "spongent: A lightweight hash function," in *Cryptographic Hardware and Embedded Systems—CHES 2011*, B. Preneel and T. Takagi, Eds. Berlin, Germany: Springer, 2011, pp. 312–325.
[32] G. Di Natale, M. Doulcier, M.-L. Flottes, and B. Rouzeyre, "Self-test techniques for crypto-devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 2, pp. 329–333, Feb. 2010.
[33] *Nexys 4 DDR [Reference.Digilentinc]*. Accessed: Aug. 2020. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start
[34] Xilinx. (2010). *7 Series FPGAs Data Sheet: Overview (DS180)*. [Online]. Available: www.xilinx.com



GERMAN CANO-QUIVEU received the B.Sc. degree in computing engineering from the University of Seville, Spain, in 2015, where he is currently pursuing the Ph.D. degree with the Department of Electronics Technology. His research interests include bootloaders, the IoT, and SoC.

PAULINO RUIZ-DE-CLAVIJO-VAZQUEZ received the B.Sc. and Ph.D. degrees in computer science from the University of Seville, Spain, in 1999 and 2007, respectively. He was with the Institute of Microelectronics, Seville, part of the National Centre of Microelectronics, Spain, from 1998 to 2004. He has been with the Department of Electronics Technology, University of Seville, since 1999, as an Assistant Professor. His research interests include system-on-chip designs, digital signal processing, and embedded microprocessors architecture, areas to which he has contributed in international conferences and workshops.

MANUEL J. BELLIDO-DIAZ received the B.Sc. and Ph.D. degrees in physics from the University of Seville, Spain, in 1987 and 1994, respectively. He has been with the Department of Electronics Technology, University of Seville, since 1990, where he holds a post as a Professor.

DAVID GUERRERO-MARTOS received the B.Sc. and Ph.D. degrees in computer engineering from the University of Seville, Spain, in 2000 and 2012, respectively. Since 2002, he has been working as a Lecturer with the Department of Electronics Technology, University of Seville. He has published several papers in journals and conferences. His research interests include digital circuit synchronization, hardware implementation of numerical methods, and computer architecture.

JULIAN VIEJO-CORTES received the M.Sc. and Ph.D. degrees in computing engineering from the University of Seville, Spain, in 2004 and 2011, respectively. He currently works as an Assistant Professor with the Department of Electronics Technology, University of Seville, and has contributed several research papers to international journals and conferences in the area of digital signal processing and system-on-chip design.

JORGE JUAN-CHICO received the B.Sc. and Ph.D. degrees in physics from the University of Seville, Spain, in 1994 and 2000, respectively. He is currently an Associate Professor with the Department of Electronics Technology, University of Seville, where he is leading the Digital Research and Development Group. He has carried out research in the areas of metastability, delay modeling, timing and power simulation, and digital embedded systems.

• • •

5.2. Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security

5.2.1. Breve resumen

En esta publicación se presenta un Core que provee confidencialidad, integridad y autenticación de datos del usuario para dispositivos IoT. Esta solución nombrada E-LUKS esta basada en el sistema de *Full Disk Encryption* utilizado en Linux (Linux Unified Key System) LUKS [FB18]. Nuestra alternativa modifica la estructura interna así como los algoritmos criptográficos usados en LUKS, para conseguir el menor uso de recursos del dispositivo.

5.2.2. Datos Revista

- Nombre Revista: MDPI Electronics
- Índice JCR(2020): 2.397(Q3)
- Fecha publicación: 5-12-2021
- DOI: https://doi.org/10.3390/electronics10233036

Article

Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security

German Cano-Quiveu *, Paulino Ruiz-de-clavijo-Vazquez, Manuel J. Bellido, Jorge Juan-Chico, Julian Viejo-Cortes, David Guerrero-Martos and Enrique Ostua-Aranguena

Department of Electronics Technology, E.T.S. Ingeniería Informática, University of Seville, Avda. Reina Mercedes s/n, 41012 Seville, Spain; pruiz@us.es (P.R.-d.-c.-V.); bellido@dte.us.es (M.J.B.); jjchico@dte.us.es (J.J.-C.); julian@us.es (J.V.-C.); guerre@dte.us.es (D.G.-M.); ostua@dte.us.es (E.O.-A.)
* Correspondence: germancq@dte.us.es

Abstract: The Internet of Things (IoT) security is one of the most important issues developers have to face. Data tampering must be prevented in IoT devices and some or all of the confidentiality, integrity, and authenticity of sensible data files must be assured in most practical IoT applications, especially when data are stored in removable devices such as microSD cards, which is very common. Software solutions are usually applied, but their effectiveness is limited due to the reduced resources available in IoT systems. This paper introduces a hardware-based security framework for IoT devices (Embedded LUKS) similar to the Linux Unified Key Setup (LUKS) solution used in Linux systems to encrypt data partitions. Embedded LUKS (E-LUKS) extends the LUKS capabilities by adding integrity and authentication methods, in addition to the confidentiality already provided by LUKS. E-LUKS uses state-of-the-art encryption and hash algorithms such as PRESENT and SPONGENT. Both are recognized as adequate solutions for IoT devices being PRESENT incorporated in the ISO/IEC 29192-2:2019 for lightweight block ciphers. E-LUKS has been implemented in modern XC7Z020 FPGA chips, resulting in a smaller hardware footprint compared to previous LUKS hardware implementations, a footprint of about a 10% of these LUKS implementations, making E-LUKS a great alternative to provide Full Disk Encryption (FDE) alongside authentication to a wide range of IoT devices.

Keywords: LUKS; embedded systems; field programmable gate array; IoT

Citation: Cano-Quiveu, G.; Ruiz-de-clavijo-Vazquez, P.; Bellido, M.J.; Juan-Chico, J.; Viejo-Cortes, J.; Guerrero-Martos, D.; Ostua-Aranguena, E. Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security. *Electronics* **2021**, *10*, 3036. https://doi.org/10.3390/electronics10233036

Academic Editors: Juan Antonio López Ramos, Antonio David Escobar Molero and José Antonio Álvarez Bermejo

Received: 12 November 2021
Accepted: 3 December 2021
Published: 5 December 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The Internet of Things (IoT) industry has grown steadily in the last few years. Many facts indicate that this growth is an upward trend [1], with IoT data traffic expected to reach around 2000 petabytes of information by 2024 [2]. The use of IoT devices has reached many fields such as industrial production [3], health care [4], or quality-of-life-related devices [5]. Its implementations in our homes and personal environments has had a great impact on many daily life processes [6,7]. However, IoT is still a recent technology in which numerous devices with low resources share a large amount of personal and sensible data, making data security one of the major issues of IoT. The security problem in IoT has been addressed by many authors, mainly from the perspective of data communication between nodes [8,9]. However, the security of the data stored internally by these devices should also be addressed.

Currently, many IoT nodes based on complex embedded systems use a Flash memory as their main storage for user applications. Some of these systems can expand their storage by adding external Flash mass storage devices such as an SD card. If the application requires some kind of confidentiality in the stored data, some type of data encryption must be used. Data encryption can be implemented at the operating system (OS) level by using dedicated libraries and factory software in order to encrypt single files or the complete block device that holds the file system. Solutions such as Bitlocker from Microsoft, FileVault from Apple, or VeraCrypt, among others, can be used at the user level by running software

tools in the OS. But these kinds of solution have to face challenges such as key storage, key change, and interoperability with other systems; for example, when the Flash storage device is removable and needs to be accessed from a desktop computer.

In any case, the presence and evolution of IoT security must be studied in order to use the available knowledge. Only in doing so can the way new devices and application areas evolve be decided [10–13].

Implementing file encryption is not an option for many IoT devices. On the contrary, block device encryption adds a new layer between the physical device and the read/write device operations. This is much more resource-friendly and allows IoT applications to access encrypted memory in a transparent way. It can pose a solution even for a very simple IoT device that does not run an OS but a standalone application. In fact, there are already solutions that implement this kind of Full Disk Encryption (FDE), such as the Linux Unified Key System (LUKS) [14].

Another important aspect of IoT security is that IoT nodes must be robust against the use of reverse engineering [15,16] and/or malware such as the Mirai botnet [17]. Thus, it is necessary to ensure the integrity and authenticity of user data in addition to its confidentiality.

In this paper, a lightweight IoT protocol based on LUKS is introduced, which provides, on top of the FDE of LUKS, both the integrity, and authenticity of the user data. This protocol has been called Embedded LUKS (E-LUKS) and it has been especially tailored to allow an efficient hardware implementation, making possible the implementation of a small footprint hardware module that is able to read/write into a final storage device. In this sense, the hardware E-LUKS module implemented will act as a transparent layer between embedded systems and their storage memories.

This paper has been organized as follows. First, the Related Work section describes other hardware solutions that can also provide security of the data. Then, in the Section titled Linux Unified Key Setup, the LUKS specification used as blueprints for the proposed solution, necessary to understand it, is described. Next, the section named Embedded LUKS details the proposed solution and its changes compared to LUKS, finishing with the E-LUKS core hardware implementation. The Results section follows, in which the experiment conducted to verify the proposed solution is represented, displaying the execution time of the system with E-LUKS and without it. Furthermore, this section also proves the advantages of E-LUKS via the comparison, and describes the subsequent analysis of the employ of resources between E-LUKS and the alternatives presented in the Related Work section. Lastly, a brief evaluation of the proposed solution is provided in Conclusions.

2. Related Work

LUKS is a block-device encryption specification that aims to be implemented in software, although there are some hardware implementations on FPGA chips as well [18,19]. In Reference [18], the implementation is focused on an energy-efficient LUKS design that can compete in terms of speed against software solutions on higher-end devices. Alternatively, there is another LUKS implementation, which achieves high performance due to a pipeline implementation, proposed in [19]. These designs are not suited for resource constrained devices, since both use multiple instances of a cryptographic algorithm implementation in order to increase the performance at the expense of additional FPGA resources.

Other hardware-solutions such as Sancus [20] are specifically designed for resource constrained devices. Sancus is an open-source solution that focuses on the integrity and authenticity of the data. Its objective is to assure that each file or program on the memory device is untampered and authenticated without trusting any infrastructural software. For this purpose, the device uses a symmetric key to assure the integrity and authenticity of each file independently by means of a Key Derivation Function (KDF) implemented in the hardware. The KDF, together with the device key and file parameters, such as its contents and location on the memory device, generates a digest for each file.

To isolate each in the device; Sancus uses a variation of the program-counter based memory access control [21]. It allows access to the protected data of the file if and only if the program counter is in its text section. In addition, the text section of a file can only be executed if the program counter jumps to its defined entry point. A call to another file can only be executed if a digest of the file to be accessed has been previously deployed in the memory device.

Based on Sancus, several additional solutions have appeared. One of them is Soteria [22], which adds confidentiality to the previous solution, and a specific software loader module. This module is responsible for the confidentiality of the other modules in the node.

Other solutions are those that implement confidentiality on a System-on-Chip (SoC) FPGA RAM memory. In Reference [23], the authors present a transparent encryption/decryption hardware module that uses block ciphers to provide confidentiality of the data. It is also able to achieve authenticity of the data by implementing a Tamper Evident Counter (TEC) tree with a nonce value as a root stored on-chip. This TEC serves to provide a nonce value, which is used only when the block cipher is the Authenticated Encryption (AE) cipher Ascon. Another proposed solution is Atlas [24], which offers confidentiality alone. The idea of Atlas is to use the entry point of the file or code as the Initialization Vector (IV) employed in the encryption/decryption of that file. Currently, the lightweight SIMON block cipher is used.

The proposed solution in this paper, E-LUKS, provides FDE as the LUKS solutions do in [18,19]. Nevertheless, as opposed to LUKS, E-LUKS has been designed for constrained resources devices and also supports data authentication in addition to confidentiality. While the solutions based on Sancus [20] require modifying the processor itself and its own firmware, the E-LUKS solution is independent of the processor and, in fact, it can be used in systems with no processor at all. Sancus-based solutions allow for the isolation of different programs in the external RAM memory of the device. In contrast, E-LUKS uses external Flash storage, which may be present as the main or complementary data storage of the system, such as a microSD card, which is usually more vulnerable to data tampering. The solutions in [23] and Atlas [24] also focus on securing the RAM memory of the system. Atlas implements instructions for the selected processor but does not support the integrity and authentication of the data. Finally, the work presented in [23] depends only on the bus interfaces of the system. However, to allow integrity and authentication of the data, it requires part of the TEC tree in the memory alongside the data, excluding the root nodes, which may consume a significant amount of memory resources.

3. Linux Unified Key Setup

Linux Unified Key Setup (LUKS) is a specification intended to standardize cryptographic key setup for data storage encryption. LUKS was introduced in 2005, and a new version (LUKS2) [25] was released in 2018. This last version extends the previous one, taking all its basic concepts as blueprints. The structure and operations described below are taken from LUKS1.

LUKS performs full user data encryption using a master key that is itself encrypted and stored in front of the encrypted data. The master key can be stored multiple times using different user passwords, allowing many users to have access to the encrypted data.

An LUKS formatted block device consists of two parts: (1) a small unencrypted part that consists of an LUKS header followed by several slots for each granted user; (2) an encrypted part starting with eight slots, each one containing the encrypted master key for one possible user, followed by the user data. All encryption is performed using symmetric cryptography.

To retrieve the encrypted data, a user must unlock one of the eight slots and decrypt the corresponding encrypted master key. A formatted LUKS device must have at least one active slot with the corresponding encrypted master key.

3.1. Cryptography

The type of cryptography used is a key factor in any secure specification. Modern cryptography is mainly supported by ciphering algorithms and cryptographic hash functions. The LUKS specification allows for the use of a variety of block ciphers and hash functions. Possible block ciphers that can be used with LUKS are shown in Table 1, together with supported modes, key lengths, and block lengths. Supported hash functions are listed in Table 2. In addition, LUKS uses Password-Based Key Derivation Function 2 (PBKDF2) as the Key Derivation Function (KDF), following the recommendations of RFC8018 [26].

Table 1. Linux Unified Key Setup (LUKS) block cipher algorithms.

Cipher	Key Lenght (bits)	Block Lenght (bits)	Modes
AES [27]	128–256	128	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64
Twofish [28]	128–256	128	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64
Serpent [29]	128–256	128	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64
cast5 [30]	128–256	128	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64
cast6 [31]	40–128	64	ecb, cbc-plain, cbc-essiv: *hash*, xts-plain64

Table 2. LUKS hash functions.

Hash	Output Lenght (bits)
sha1 [32]	160
sha256 [33]	256
sha512 [33]	512
ripemd160 [34]	160

3.2. Linux Unified Key Setup Internal Layout

As previously mentioned, the layout of an LUKS block device is divided into two parts, one is unencrypted and the other one is encrypted, as depicted in Figure 1. The unencrypted part is called the LUKS partition header (*LUKS phdr*). It starts at the beginning of the block device and holds two types of blocks: the *header* and various Key Slots (KS_x). The encrypted part contains two types of blocks as well: Key Materials (KM_x) blocks and *User Data* blocks. The *header* identifies the block device as an LUKS partition and stores information about the master key, together with information about the cryptographic algorithms selected in the LUKS block device. Table 3 summarizes the fields of the *phdr*. The Key Slots (KS_x) have information about the user key and the parameters to recover the master key. The 8 KS_x allow up to eight users to access the master key, each one with a different personal user password. Each of the KS_x blocks ($x = 1, 2, \ldots, 8$) points to a KM_x block, which is the encrypted master key that can only be recovered using the corresponding personal user password. Table 4 shows all fields of a KS_x. Following the KM_x blocks are the *User Data*, which are encrypted with the master key.

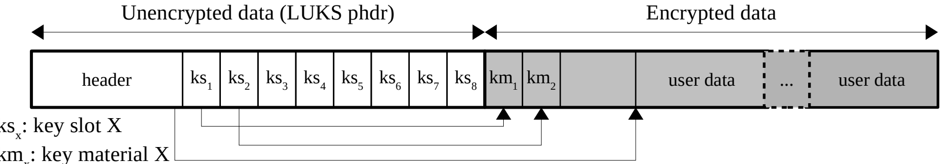


Figure 1. LUKS layout.

Table 3. LUKS partition header (LUKS phdr) fields.

Offset (Bytes)	Field Name	Lenght (Bytes)	Description
0	magic	6	indicates an LUKS partition
6	version	2	indicates LUKS version
8	cipher-name	32	String with the cipher used
40	cipher-mode	32	String indicating the cipher mode
72	hash-spec	32	String with the hash used
104	payload-offset	4	block (512-bytes sector) where the encrypted data begins
108	key-bytes	4	lenght of the master key
112	mk-digest	20	master key digest from Key Derivation Function (KDF)
132	mk-digest-salt	32	salt parameter for master key in KDF
164	mk-digest-iter	4	count parameter for master key in KDF
168	uuid	40	UUID of the partition
208	KS_1	48	Key Slot 1
…	….	…	…
544	KS_8	48	Key Slot 8

Table 4. LUKS Key Slot (KS_x) fields.

Offset (Bytes)	Field Name	Lenght (Bytes)	Description
0	active	4	indicates if the KM_x is enabled
4	iterations	4	count parameter for the KDF
8	salt	32	salt parameter for the KDF
40	key-material-offset	4	block (512-bytes sector) where the KM_x begins
44	stripes	4	number of anti-forensic stripes

3.3. *Operations*

LUKS devices are managed using four types of operation: *initialisation*, *add new password*, *master key recover*, and *password revocation*. By using these functions, LUKS is able to store and retrieve the necessary master key to perform any operation on disk. These functions are summarized below.

3.3.1. *Initialisation*

The *initialisation* operation consists of formatting the block device according to the LUKS layout. This operation requires the following parameters: the master key, the salt for the KDF (*mk-digest-salt*), and a number of iterations (*mk-digest-iter*). These parameters are auto-generated by the software that performs the LUKS formatting. In the process, a new *LUKS phdr* is written in the block device, followed by KS_x. It is also mandatory to specify which cryptographic algorithms (block cipher and hash function) are selected to be used on the block device. To complete the *LUKS phdr*, the master key is used as a parameter, along with the salt and the number of iterations, to generate a digest (*mk-digest*). Lastly, the LUKS format must store the encrypted master key in at least one KM_x. Because the master key is encrypted by a user key, a new user password needs to be added in the way described below.

3.3.2. *Add New Password*

This operation consists of adding a new password for a user, using one of the eight KS_x available. When the block device is being formatted, the unencrypted master key is available from the *initialisation* operation. On the contrary, for an already LUKS-formatted device, the master key is retrieved by the *Master Key Recovery* operation. The *Add New Password* operation begins with the user providing a new user password. After that, it generates random values from the salt and iteration count for the KS_x and stores them. Once the values are generated, the KDF takes the new password along with the salt and the iteration count. The master key is then processed by an anti-forensic splitter, generating a new derived key. This derived key is encrypted using the KDF output as the key for the cipher. Then, the encrypted password is stored on the device as KM_x for later use.

3.3.3. *Master Key Recovery*

The third operation is to recover the master key from a KM_x. This operation consists of decrypting the encrypted master key for two purposes: access to the encrypted user data or to change a user password by using the *Add New Password* operation. The *Master Key Recovery* operation requires the user password associated with the KS_x. It uses the KS_x salt and iteration count values, along with the user password, as parameters for the KDF. Next, the KM_x is recovered from the storage device and is decrypted using the KDF output as the cipher key. Then, the decrypted result is processed by an anti-forensic merge, generating a new candidate key. This candidate key, the *mk-digest-salt* and the *mk-digest-iter* are passed as parameters to the KDF. Finally, the KDF output is compared with the *mk-digest* from the *header*, and if both values match, the candidate key is returned.

3.3.4. *Password Revocation*

The last operation is the password revocation for a selected key slot. This operation consists of deleting the selected KM_x and setting the *activate* field of the KS_x to inactive.

4. Embedded LUKS

Embedded LUKS (E-LUKS) is a new proposal to bring LUKS to devices with limited resources, such as IoT devices. It can be applied in scenarios where the security of the local data is an issue, such as devices in public locations, where an attacker has easy access to the device. E-LUKS performs FDE of the block device. To perform this task, it requires a master key, which is used to create the user keys, as well as in LUKS. The difference being that E-LUKS allows integrity and authentication, on top of confidentiality, of the user data.

Another major aspect to consider is the cryptographic algorithm used for small devices with limited resources. In the last few years, a growing trend of low footprints cryptographic algorithms has emerged for these types of devices. Although these algorithms are designed mainly to optimize the use of internal device resources, at the same time, they also decrease the security level. Nevertheless, the security provided by these algorithms is enough and presents more advantages than disadvantages.

4.1. E-LUKS Internal Layout

The E-LUKS layout is heavily based on LUKS. In Figure 2, a diagram of the layout is shown. However, in E-LUKS, the Key Material (KM_x) is integrated into the Key Slot (KS_x). In addition, some parts present differences in their number of fields, as can be seen in Tables 5 and 6 with respect to the *E-LUKS header* and KS_x. Another characteristic of E-LUKS is that the *E-LUKS header* and the KS_x must be located in the first 512 bytes of the memory. This limitation is due to the fact that embedded devices usually have an external flash memory, such as a microSD card, as their main storage. These flash memories are typically divided into blocks of 512 bytes and, to facilitate the HDL design, it has been decided that the first block of the memory contains all the E-LUKS related data, while the rest of the memory is reserved for the user encrypted data. The fields below are taken by selecting the PRESENT cipher with an 80-bit key and, as a hash function, the same as that

used in the KDF and the HMAC, the SPONGENT-88. With these requirements, the *E-LUKS header* takes 52 bytes, and each KS_x takes 40 bytes. Therefore, the maximum number of KS_x is up to 11.

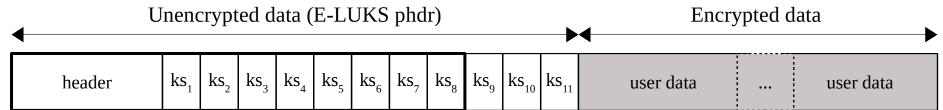


Figure 2. Embedded LUKS (E-LUKS) layout.

Table 5. E-LUKS header fields.

Offset (Bytes)	Field Name	Lenght (Bytes)	Description
0	magic	6	indicates an E-LUKS partition
6	mk-digest	11	master key digest from KDF
17	mk-digest-salt	8	salt parameter for master key in KDF
25	mk-digest-iter	4	count parameter for master key in KDF
29	mk-hmac	11	output from the HMAC of the encrpyted user data
40	mk-IV	8	Initialization Vector (IV) parameter for the block cipher
48	user-data-blocks	4	blocks (512-bytes sector) of user data

Table 6. E-LUKS Key Slot (KS_x) fields.

Offset (Bytes)	Field Name	Lenght (Bytes)	Description
0	activate	4	indicates if the KS_x is enabled
4	iterations	4	count parameter for the KDF
8	salt	8	salt parameter for the KDF
16	pwd-encrypted	16	key material KM_x, encrypted master key
40	IV	8	IV parameter for the block cipher used to generate pwd-encrypted

4.2. Cryptographic Algorithms

The cryptography used in ELUKS inherits from LUKS the way to store the master key. In this sense, ELUKS uses a block cipher, a HASH function, and a KDF. The main feature of E-LUKS is the fact that it uses cryptographic algorithms designed for resource constrained devices. To simplify the design and standardize the internal layout, it has been decided that there will be only one choice for each of the algorithms (block cipher, hash function, HMAC, and KDF). These algorithms are introduced in the following sections.

4.2.1. PRESENT

PRESENT is a block cipher optimized for low resource usage that also has low power consumption. In recent years, multiple implementations of this cipher have appeared with different hardware architectures to boost certain aspects [35]. However, even though there are multiple alternatives for PRESENT, this work has implemented it as was originally

presented in [36]. PRESENT is a substitution–permutation network (SP-network) of 31 rounds of encryption–decryption, all of them with the same structure, which uses a block length of 64 bits and allows keys of 80 bits and 128 bits. Its pseudocode is shown in Algorithm 1.

Algorithm 1 Pseudo-code of the PRESENT encrypt operation.

1: $STATE \leftarrow 0$
2: $ROUNDKEYS = [K_1, K_2, \ldots, K_{32}]$
3: $ROUNDKEYS \leftarrow generateRoundKeys()$
4: **for** $i \leftarrow 1$ to 31 **do**
5: $\quad STATE \leftarrow addRoundKey(STATE, K_i)$
6: $\quad STATE \leftarrow sBoxLayer(STATE)$
7: $\quad STATE \leftarrow pLayer(STATE, K_i)$
8: $STATE \leftarrow addRoundKey(STATE, K_{32})$
9: **return** $STATE$

Each of the functions of the algorithm is described as follows:

- *addRoundKey*: Given the *round key* $K_i = k_{63}, k_{62}, \ldots, k_0$ for $1 \leq i \leq 32$ and the current $STATE = b_{63}, b_{62}, \ldots, b_0$, the function returns a new state $RESULT = r_{63}, r_{62}, \ldots, r_0$, calculated as:
$$r_j = b_j \oplus k_j. \quad \text{for} \quad 0 \leq j \leq 63. \tag{1}$$
- *sBoxLayer*: Is a function with an input of 64 bits that returns an output of 64 bits. Internally, the input data are divided into 16 groups of 4 bits, such that $INPUT = b_{63}, b_{62}, \ldots, b_0 = w_{15}, w_{14}, \ldots, w_0$. Each group ($w_i$) is then processed by *S-boxes*, an S-box S transforms 4-bit input into 4-bit output such that $S : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$. Table 7 shows all possible *S-boxes* results. Finally, the *sBoxLayer* and the result are calculated as follows:
$$\left.\begin{aligned} w_i = b_{4*i+3} \parallel b_{4*i+2} \parallel b_{4*i+1} \parallel b_{4*i+0} \\ RESULT = S[w_{16}], \ldots, S[w_0] \end{aligned}\right\} \quad \text{for} \quad 0 \leq i \leq 15.$$

Table 7. PRESENT *S-box*.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

- *pLayer*: The pLayer operation performs a permutation operation defined by:
$$P(i) = \begin{cases} i * 16 \mod 63, & i \in 0, \ldots, 62 \\ 63, & i = 63. \end{cases}$$
- *generateRoundKeys*: Let us assume a length key of 80 bits such as $K = k_{79}, k_{78}, \ldots, k_0$. It is necessary to generate a different round key K_i of 64 bits for each of the rounds. The steps are the following:
$$1.\ [k_{79}, k_{78}, \ldots, k_1, k_0] = [k_{18}, k_{17}, \ldots, k_{20}, k_{19}] = K << 61 \tag{2}$$
$$2.\ [k_{79}, k_{78}, k_{77}, k_{76}] = S[k_{79}, k_{78}, k_{77}, k_{76}] \tag{3}$$
$$3.\ [k_{19}, k_{18}, k_{17}, k_{16}, k_{15}] = [k_{19}, k_{18}, k_{17}, k_{16}, k_{15}] \oplus i \tag{4}$$
$$4.\ K_i = k_{79}, k_{78}, \ldots, k_{16} = MSB_{64}(K). \tag{5}$$

Block ciphers can operate in different modes, the default mode being the ECB mode (Electronic Code Book mode). However, this mode does not provide security as the other modes do. Therefore, for the PRESENT cipher in this work, it has been decided to select the

CTR mode (counter mode). In this mode, the cipher behaves as a stream cipher, thus being used to generate a keystream s. In the equation below, the CTR mode is described where m_i is the i-th 64-bit unencrypted block and c_i is the i-th encrypted block. In this mode, x_1 begins at a random value IV (Initialization Vector) and behaves as a counter. Therefore, this mode allows each encrypted block to be different, regardless of its data.

$$\left.\begin{aligned} c_i &= m_i \oplus MSB_s(ENC_k(x_i)) \\ x_{i+1} &= INC(x_i) \end{aligned}\right\} \quad \text{for} \quad 1 \leq i \leq n.$$

4.2.2. SPONGENT

SPONGENT [37] is a hash function based on the sponge architecture and the use of the permutation operation introduced in PRESENT.

The sponge architecture is an iterative design composed of stages. It takes an arbitrary length of input data and generates an output whose length is related to the number of stages in the architecture. The parameters and operations are the following:

- r: bits length of the ratio.
- c: bits length of the capacity.
- R: number of rounds that take place in each stage.
- n: bits length for the output of the sponge architecture.
- b: bits length for the internal state, which is $b = r + c$.
- π_b: represents the function for the permutation operation. This function $\pi_b(x) = y$ such that $x, y \in 0, 1^b$. In addition, this function is the main operation in each stage of the sponge architecture.

Once the parameters are defined, it is necessary to describe the three phases of the sponge architecture.

- *Initialization phase*: This phase pads the input M with a '1' followed by many '0' until achieving $len(M) \mod r = 0$. Then, the input M is divided into blocks of r-bits such that $M = m_1, m_2, \ldots m_{\frac{len(M)}{r}}$.
- *Absorbing phase*: This phase is composed of stages. In each stage, the input block m_i is added to the current state, as follows $STATE = STATE \oplus m_i$. Once the input is added, a new state is generated by the permutation operation such that $STATE = \pi_b(STATE)$.
- *Squeezing phase*: This phase, as well as the absorbing phase, is composed of stages. Each stage generates r-bits of the output h, from the state $h_i = STATE[b-1 : b-1-r] = MSB_r(STATE)$. Then, a new state is created $STATE = \pi_b(STATE)$. The stages go on until n-bits of the output h have been generated.

SPONGENT offers five different variants to reach different levels of security. Variants, as well as the value of the parameters, are shown in Table 8.

Table 8. Security levels of SPONGENT.

Variant	*n*	*b*	*c*	*r*	*R*	*Preimage*	*2nd Preimage*	*Collision*
SPONGENT-88	88	88	80	8	45	80	40	40
SPONGENT-128	128	136	128	8	70	128	64	64
SPONGENT-160	160	176	160	16	90	144	80	80
SPONGENT-224	224	240	224	16	120	208	112	112
SPONGENT-256	256	272	256	16	140	240	128	128

PRESENT Permutation π_b

The permutation operation π_b is the main part of the design. The behaviour of the function can be defined as follows:

for $(i = 1; i \leq R; i = i + 1)$
$STATE = lCounter_b(i)[0 : \log_2(R) - 1] \oplus STATE[b - 1 : b - 1 - \log_2(R)]$
$STATE = lCounter_b(i)[\log_2(R) - 1 : 0] \oplus STATE[\log_2(R) - 1 : 0]$
$STATE = sBoxLayer_b(STATE)$
$STATE = pLayer_b(STATE)$
end for.

The $sBoxLayer_b$ and $pLayer_b$ functions are based on their homology described in the PRESENT section. The main difference is that these functions are modified to accept either input or output of b-bit. In the case of the $sBoxLayer_b$, more S-boxes have been used in parallel. Thus, $pLayer_b$ is defined as:

$$pLayer_b(i) = \begin{cases} i * (b/4) \mod b - 1, & i \in 0, \dots, b - 2 \\ b - 1, & i = b - 1. \end{cases}$$

Regarding the $lCounter_b$, it is an LFSR of $\log_2(R)$-bits. This register is activated in each iteration $i = 1, 2, \dots, R$. It uses different irreducible polynomials as coefficients and different initial values for each variant of SPONGENT, as shown in Table 9.

Table 9. $lCounter_b$ values.

Variant	Polynomial	Initial Value
SPONGENT-88	$x^6 + x^5 + 1$	$0x05$
SPONGENT-128	$x^7 + x^6 + 1$	$0x7A$
SPONGENT-160	$x^7 + x^6 + 1$	$0x45$
SPONGENT-224	$x^7 + x^6 + 1$	$0x01$
SPONGENT-256	$x^8 + x^4 + x^3 + x^2 + 1$	$0x9E$

HMAC

The hash function is used in E-LUKS as part of the HMAC construction. An HMAC allows both: integrity and authentication. The HMAC used was presented in [38] with the following definition:

$$HMAC_K(X) = Y.$$

First, the key K is XORed with a repetitive pattern of bits called $ipad$. Then, m_0 is calculated as the hash of the input of the HMAC with a suffix of the XORed key.

$$ipad = 0x33, 0x33, \dots, 0x33$$

$$m_0 = h[(K \oplus ipad)||X].$$

After that, it proceeds to calculate the output of the HMAC. To this end, another repetitive pattern of bits called $opad$ is XORed with the key and is used as a suffix with the previous hash output m_0. These data are then passed to the hash function to get the output:

$$opad = 0x5C, 0x5C, \dots, 0x5C$$

$$Y = h[(K \oplus opad)||m_0].$$

4.2.3. KDF

The KDF used in E-LUKS is based on the PBKDF2, the KDF used in LUKS. Therefore, this function will be explained.

$PBKDF2(P, S, c, dkLen)$ contains the following parameters:

- P: the master key.

- S: *salt* value.
- c: represents the number of iterations.
- $dkLen$: length of the derivated key.

First, $PBKDF2$ checks that the desired derived key is in the range in which $hLen$ represents the output length of the hash function.

$$dkLen > (2^{32} - 1) * hLen \rightarrow \textbf{ERROR}.$$

Then, it calculates the number of blocks of length $hLen$ used in the derivated key. This value is stored and, for the last block, in the case of a non-integer result dividing by $hLen$, the number of bytes is also stored.

$$l = \left\lceil \frac{dkLen}{hLen} \right\rceil \tag{6}$$

$$r = dkLen - [(l-1) * hLen]. \tag{7}$$

For each of the blocks of length $hLen$, T_i, the F function will be applied, which takes P, S, c and the index block as a parameter.

$$T_1 = F(P, S, c, 1) \tag{8}$$

$$T_2 = F(P, S, c, 2) \tag{9}$$

$$\ldots \tag{10}$$

$$T_l = F(P, S, c, l). \tag{11}$$

The function F is the XOR of the first c iterations of the HMAC function H.

$$F(P, S, c, i) = U_1 \oplus U_2 \oplus \cdots \oplus U_c \tag{12}$$

$$U_1 = HMAC_P(S||i) \tag{13}$$

$$U_2 = HMAC_P(U_1) \tag{14}$$

$$\ldots \tag{15}$$

$$U_c = HMAC_P(U_{c-1}). \tag{16}$$

Lastly, all the calculated blocks are concatenated in order to generate the derivated key DK.

$$DK = T_1||T_2||\ldots||T_l[r-1:0].$$

However, this construction could be improved as shown in [39]. This has been the alternative chosen for E-LUKS.

In this construction, instead of an HMAC, a hash function was used for the F function. In addition, the parameter c is passed as a parameter for the hash function.

$$y = F(P, S, c) = H^c(P||S||c).$$

Here, H^c represents the hash function executed c times as follows:

$$y_c = H^c(P||S||c)$$

$$y_1 = H(P||S||c)$$

$$y_2 = H(y_1)$$

$$\ldots$$

$$y_c = H(y_{c-1}).$$

4.3. Operations

Similarly to the previous subsection, E-LUKS has the same operations as LUKS, adding a new one that allows the integrity and authentication of the user data.

4.3.1. Initialisation

The main difference in this operation compared to LUKS is the inclusion of integrity and authentication of the user data. Therefore, to allow integrity and authentication, this stage requires the calculation of the HMAC of all the encrypted user data. Below is a brief description of the whole operation.

The *initialisation* operation requires the master key, *mk-digest-iter*, *mk-digest-salt* and *mk-IV*. It can be divided into three parts; the first one populates the simple fields of the *E-LUKS header* as: the random value for the salt of the KDF, the constant value of the magic field, the count value, and the initial value. The second part generates the KDF of the master key with the salt and count parameters. Lastly, the third part generates the HMAC digest of all the encrypted user data.

4.3.2. *Add New Password*

The most significant change in this operation with respect to LUKS is the exclusion of the anti-forensic functions. Therefore, there is no need to calculate the split key.

The *Add New Password* operation requires the master key, the count iteration value for the KS_x, and the user key. This operation begins by setting the selected KS_x to activate. Doing so, random values start to generate for the salt and Initial Value fields. Once all the values have been generated for the KS_x, the user key is processed by the KDF given the user key digest. Then, the master key will be divided into chunks the size of the block cipher size. Each chunk will be encrypted with the user key digest and is concatenated to create the KM_x.

4.3.3. *Master Key Recovery*

It is similar to the *Add New Password operation*; the main difference is the exclusion of the anti-forensic functions.

The *Master Key Recovery* operation requires the user key. First of all, it performs a search for each KS_x. If the activate field is set, then it will create the user key digest with the KDF, using the parameters of the KS_x. Next, it divides the encrypted password into chunks the size of the block cipher; each chunk is decrypted using the user key digest, and each part is concatenated into the master key candidate. This master key candidate is then processed by the KDF with the parameters from the *E-LUKS header* and is compared with the *mk-digest*. If both values are equal, then the master key candidate is returned.

4.3.4. *HMAC Verification*

The *HMAC verification* is a new operation for E-LUKS, which requires the master key. Once the HMAC is initialised with the master key, the user data are then divided into chunks and are fed to the HMAC. When all the data have been fed, the HMAC generates a digest, which is compared with the *mk-hmac* stored in the *E-LUKS header*.

This operation allows for the integrity and authentication of the user data. Therefore, it brings out the possibility of creating a secure boot for the device in which E-LUKS is running.

4.3.5. *Password Revocation*

The *Password Revocation* is the last operation, an exact duplicate of its counterpart in LUKS.

4.4. Comparison between LUKS and E-LUKS

This subsection shows the difference between LUKS and E-LUKS, as shown in Table 10.

Table 10. Properties of LUKS and E-LUKS.

		LUKS	**E-LUKS**
Cryptography	block ciphers	aes, twofish, serpert, cast5, cast6	PRESENT
	block ciphers mode	ecb, cbc-plain, cbc-essiv:*hash*, xts-plain64	ctr
	hash functions	sha1, sha256, sha512, ripemd160	SPONGENT-88
	KDF	Password-Based Key Derivation Function 2 (PBKDF2)	KDF [39]
	HMAC	-	based on SPONGENT-88
Layout	header size (bytes)	208	52
	KS_x size (bytes)	48	48
	phdr size (bytes)	596	512
Operations	Initialisation	yes	yes
	Add New Password	yes	yes
	Master Key Recovery	yes	yes
	HMAC verification	no	yes
	Password Revocation	yes	yes

Before starting to describe the hardware implementation of E-LUKS, Table 11 shows the key differences between LUKS and E-LUKS. LUKS allows more security against brute-force attacks, but lacks integrity and authentication of the data. Therefore, while LUKS cannot provide a mechanism of secure boot, it is possible for E-LUKS.

Table 11. LUKS and E-LUKS comparison.

	LUKS	**E-LUKS**
confidentiality	yes	yes
integrity	no	yes
authentication	no	yes
max. cipher key lenght (bits)	256	80
max. cipher block lenght (bits)	128	64
max. digest lenght (bits)	512	88

4.5. Hardware Implementation

E-LUKS has been implemented in SystemVerilog, the schematic of which is shown in Figure 3. The E-LUKS core has four main parts, three of them related to the aforementioned cryptographic algorithms described earlier: the PRESENT cipher core, the HMAC based on SPONGENT, and the KDF. The last part is the control module, a finite state machine, which implements the Master Key Recovery operation and the *HMAC verification* operation.

In addition, the control module has an SPI interface to communicate with the formatted E-LUKS memory.

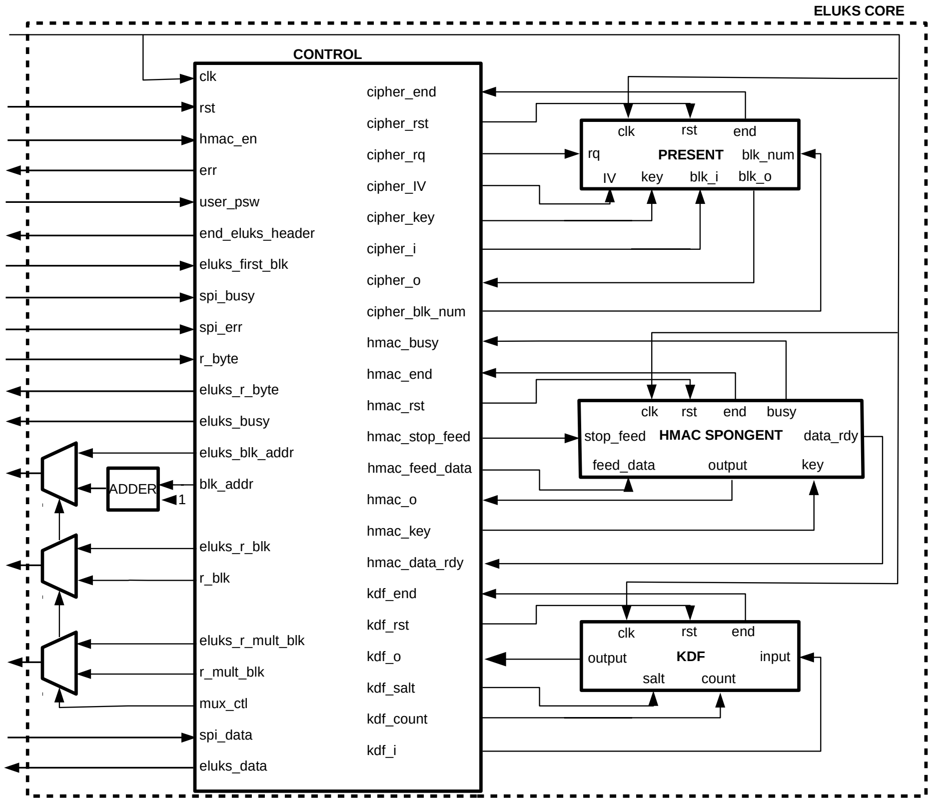


Figure 3. E-LUKS core schematic.

The flow chart that represents the *Master Key Recovery* operation performed by the control module is shown in Figure 4.

The operation begins with the reset signal. After that, the control module resets all the internal counters and registers and proceeds to read from the memory the first block of the E-LUKS partition. Then, it stores all the fields from the *E-LUKS header*. Although some of the fields are not used in this operation, they will be used later by other operations (*HMAC verification* or reading the encrypted user data from the E-LUKS partition). After that, the *magic* field is compared with the `E-LUKS_ID`, which identifies it as an E-LUKS partition. However, in another case, it reaches the *error state* from which the error signal is raised, and finishing the operation. Following the operation, the control module proceeds to read each key slot KS_x in order, checking for each one if the *activate* field is set. If all KS_x are deactivated, the control module goes to the *error state*. If the KS_x is activated, it continues to decrypt the *pwd-encrypted* field. First, it must calculate the key to decrypt, done by calculating the KDF of the *user_psw*, provided as an input of the E-LUKS core. In addition, the KDF takes the *salt* and *iterations* fields related to the KS_x. Once with the key, it is able to decrypt the *pwd-encryted* getting a master key candidate. To verify this candidate, it generates the KDF of the candidate with the fields *mk-digest-salt* and *mk-digest-iter*. Finally, the calculated digest is compared against the field *mk-digest*. If both values are equal, then the candidate is the master key and is returned. Otherwise, the control module goes to the *error state*.

Regarding the *HMAC verification* operation, this is only performed if the *hmac-enable* signal is activated. A flow chart of this operation is shown in Figure 5. The *HMAC verification* operation takes place after a successful *Master Key Recovery* operation. Therefore, it is assumed that the required fields from the *E-LUKS phdr* have been precisely stored. The operation begins by initializing the HMAC with the master key. Then, it reads the encrypted user data that begins in the second block of the E-LUKS partition. It reads

all data byte by byte, each of which feeds the HMAC. When all user data is processed, the control module calculates the HMAC output. Finally, the digest generated by the HMAC is compared to the *mk-hmac* field. If both values are equal, then the user data are proven to be authenticated and unmodified. Otherwise, the control module goes to the *error state* described earlier in the *Master Key Recovery* operation.

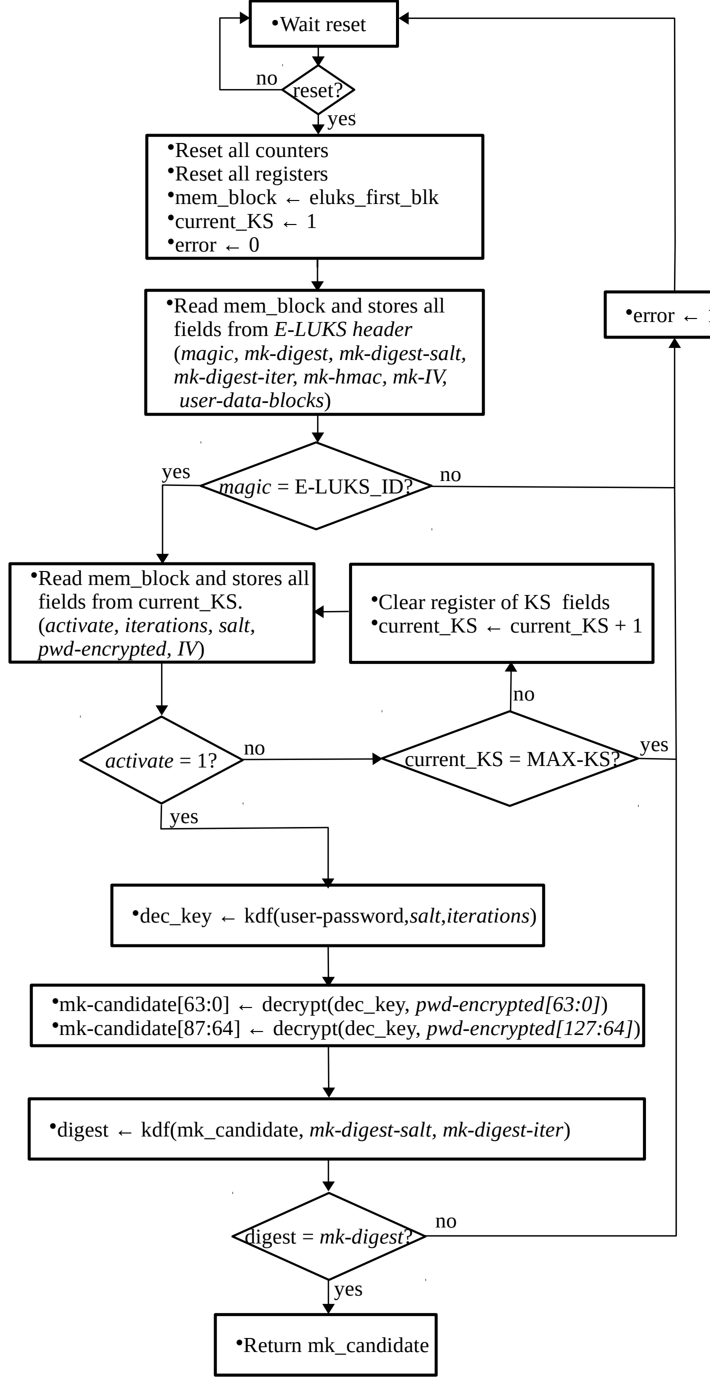


Figure 4. Flow diagram of the Master Key Recovery operation.

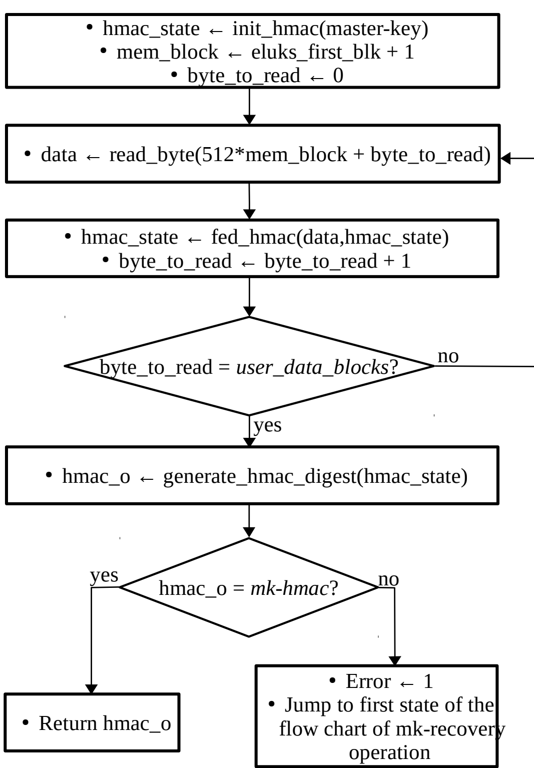


Figure 5. Flow diagram of the HMAC verification operation.

5. Results

In order to validate the proposed solution, an experiment has been conducted that provides a functional verification of E-LUKS and execution times with and without it. These resource results are then compared to the previous solution shown in Related Work.

The experiment has been designed using the SystemVerilog Hardware Description Language (HDL) and is implemented on a Nexys4DDR development board from Digilent Inc. that has an XC7A100T FPGA chip from Xilinx Inc. [40]. The selected development environment is Vivado 2020.1 from Xilinx Inc. [41]. This board features a microSD card slot.

A microSD card is the system's storage device. It is divided into two parts: the first part contains the E-LUKS partition, which stores a target file as the encrypted user data, and the second part contains the unencrypted target file.

The main goal of the experiment is to obtain the execution times when reading the target file in different ways: (1) From inside an E-LUKS partition. It can be done either without the HMAC verification as with LUKS or with the HMAC verification to provide authentication and integrity of the data; (2) From the unencrypted memory area.

To perform the experiment, three different and independent tasks have been developed:

1. Functional verification: The first task begins by reading the unencrypted target file and storing it in a block RAM in the FPGA. Once the unencrypted target file is read, the E-LUKS partition is accessed in order to get the encrypted target file. The E-LUKS can be accessed with HMAC verification or not, depending on the *hmac-enable* signal attached to any switch from the board. For its part, the encrypted file is read and compared with the previously stored values. If all data are equal, then the functional verification is correct.
2. Reading the unencrypted target file: This task reads the unencrypted target file from the microSD card. The duration of this process is measured with an internal counter that gives a precise execution time for this task. The results are displayed in a 7-segments display on the board. This execution time excludes the initialisation time for the microSD card, which takes around 250 ms.

3. Reading the target file from the E-LUKS partition: This last task retrieves the target file from the E-LUKS partition both with or without HMAC verification, depending on the value of *hmac-enable*. It starts by reading the *phdr* from the microSD card to get the master key. Once the master key is retrieved, it is stored in a volatile memory accessible only to the E-LUKS core. Thus, it is only necessary to get the master key once. As in the previous task, an internal counter is used to calculate the execution time and display it on a 7-segments display skipping the microSD card initialisation time.

All the source code is available on the authors' github [42]. The files are ready to be used with the Fusesoc tool [43], which allows the reuse of previous designs and facilitates the creation of the bitfile. This repository has three folders:

1. hdl: Contains the SystemVerilog implementation of the E-LUKS core. In addition, it also holds the core in a format that can be used with the Fusesoc tool.
2. examples: Contains the files needed to replicate the proposed experiment. The files needed to create the bitfile are in the *fusesoc* folder. Whereas the file *create_partition.py* in the *python* folder is a script that generates the layout needed in this experiment for a microSD card.
3. cores: Contains a list of Fusesoc cores used in this experiment taken from previous designs.

In the first task, the file from the unencrypted partition is read and compared with the file read from the E-LUKS encrypted partition, both with HMAC and without HMAC verification. If the comparison is successful, E-LUKS is proven to be functionally valid.

The second and third tasks are intended to obtain the execution times of the reading operations. Different sizes have been used for the target file: 4 KB, 8 KB, and 16 KB. In addition, the count value stored in the *E-LUKS header* and the KS_x, is also variable. The count value increases the security by stretching the original password length (88 bits) with $\log_2$ *count-value* [39]. Therefore, the total password length obtained for the different count values is 93-bit for 32, 94-bit for 64, and 95-bit for 128.

The execution time results from the experiment are shown in Table 12. These results, as expected, indicate that the execution time increases when the target file size or the count value increases. It is also observed that the HMAC verification is the worst case scenario for execution time, but the best case scenario from the security perspective. Therefore, in order to provide more security to the system, the payoff is the execution time. In addition, it is noticed that the percentage of time needed for the E-LUKS, compared with the unencrypted file, is decreased with larger files (+418.8% for 4 KB, +310.3% for 8 KB, and +257.1% for 16 KB).

Regarding the resource taken from the FPGA. E-LUKS has been designed to occupy the least amount of resources. To get a better perspective, the resources' results have been compared with previous solutions. To this end, the E-LUKS core has been synthesized in different FPGAs, XC7Z020 [44] and XC6VLX240T [45]. The results can be divided in two groups: (1) results for solutions that are not specifically for resource constrained devices, and (2) results for solutions that are designed for these devices.

In the first group are the LUKS hardware implementations in [18,19], already presented in the Related Work Section. Its results, shown in Table 13, are an order of magnitude above E-LUKS. It is remarkable that E-LUKS takes less than 90% of the resources when compared to these LUKS hardware implementations. In Figure 6, it is observed that the best LUKS results are those of [18], which takes around 50% of Slice LUTs and 30% of Slice Registers from the FPGA. However, E-LUKS only takes about 5% of Slice LUTs and 3% of Slice Registers. Therefore, E-LUKS proves to be better suited to performing FDE in resource constrained devices which seek to have a small footprint over time execution. In this regard, the LUKS hardware solutions offer better performance due to their pipeline implementation.

The second group is about hardware solutions specifically for resource constrained devices. It contains Sancus [20], Soteria [22], Atlas [24], and [23]. The results are shown

in Table 14 and Figures 7 and 8. All the solutions, except [23] when using authentication with the AE cipher ASCON, utilize less than 10% of any kind of resources for the FPGA. Therefore, these solutions are an adequate fit for resource constrained devices. When E-LUKS is compared to [23], it is observed that E-LUKS presents a slight improvement. Specially, when both designs are used to provide confidentiality, integrity, and authentication, whereas [23] takes up to four times more Slice LUTs. Regarding Atlas, it presents better results in Slice LUTs, 0.7% against 1.8%. However, it has worse results in number of Slices, 2.5% against 6.5%, and Slice Registers, 0.9% against 1.8%. Atlas provides confidentiality, although it lacks integrity and authenticity. Lastly, there are Sancus and Soteria. Both solutions have almost identical results, being the best suited, although it only shows when 1 SM is on the device. The overhead for an additional SM is 0.13% of Slice LUTs, and 0.013% of Slice Registers. E-LUKS takes about two times more Slice Registers and three times more Slice LUTs. However, these values represent less than 2% of both resources. With 10 SM, the Slice LUTs of E-LUKS and both solutions offer similar results. Therefore, to fit a few programs, Sancus and Soteria are the best solutions. However, to accommodate a large number of files, FDE is a better fit. Hence, to accommodate a large number of files, enough memory is required. In these kinds of devices, the storage for large files is usually a flash removable device, such as a microSD card. These memories are the target of E-LUKS, instead of the RAM memory. This leads to the advantage of being independent of any processor. It only needs to implement a bridge to the system bus, as well as [23].

Table 12. Time results for the XC7A100T FPGA.

File Size	Count Value	Unencrypted (Time ms)	E-LUKS Encrypt (Time ms)	E-LUKS Encrypt + HMAC (Time ms)
4 KB	32	1.81 (+0.0%)	4.16 (+128.9%)	6.8 (+275.7%)
	64	1.81 (+0.0%)	5.15 (+184.5%)	7.40 (+308.8%)
	128	1.81 (+0.0%)	7.15 (+295.0%)	9.39 (+418.8%)
8 KB	32	3.41 (+0.0%)	6.75 (+97.9%)	10.77 (+215.8%)
	64	3.41 (+0.0%)	7.75 (+127.3%)	11.76 (+244.9%)
	128	3.41 (+0.0%)	9.74 (+185.6%)	13.99 (+310.3%)
16 KB	32	6.58 (+0.0%)	11.89 (+80.7%)	19.90 (+202.4%)
	64	6.58 (+0.0%)	12.89 (+95.9%)	21.13 (+221.1%)
	128	6.58 (+0.0%)	14.88 (+126.1%)	23.50 (+257.1%)

Table 13. Resources comparison with LUKS solutions for the XC7Z020 FPGA.

Core	FPGA	Slice LUTs	Slice Registers	BRAMs
E-LUKS	XC7Z020	2672	2732	1
LUKS [18]	XC7Z020	28068	29866	36
LUKS [19]	XC7Z020	41656	66447	22

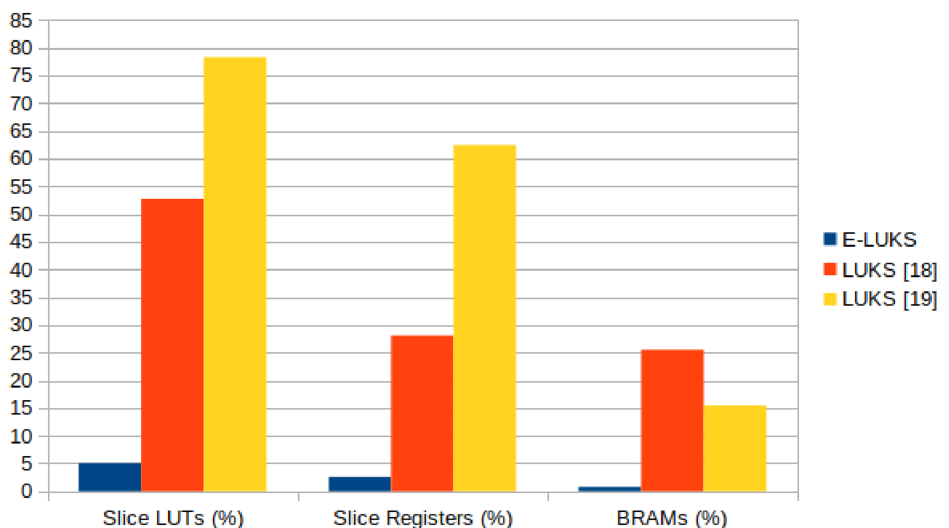


Figure 6. Percentage resources of E-LUKS and LUKS solutions for the XC7Z020 FPGA.

Table 14. Resources comparison with resource-constraint solutions for the XC6VLX240T FPGA, and XC7Z020 FPGA. The - values are not provided in the original papers.

Core	FPGA	Slice	Slice LUTs	Slice Registers	BRAMs
E-LUKS	XC7Z020	946	2672	2732	1
[23] without Authentication	XC7Z020	-	4735	2447	4.5
[23] with Authentication	XC7Z020	-	10214	4682	4.5
E-LUKS	XC6VLX240T	936	2724	2691	1
Sancus [20] with 1 *SM*	XC6VLX240T	-	751	1166	-
Soteria [22] with 1 *SM*	XC6VLX240T	-	792	1374	-
Atlas [24]	XC6VLX240T	2451	1025	5412	-

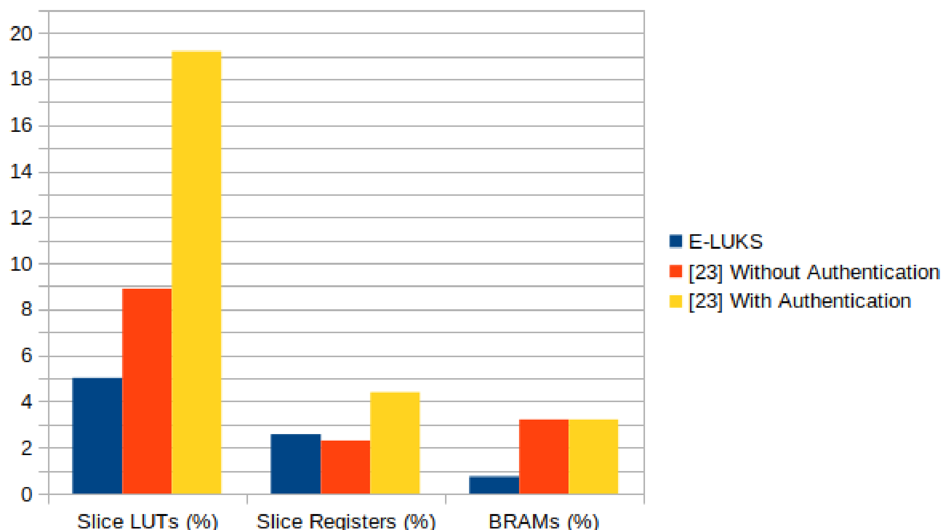


Figure 7. Percentage resources of E-LUKS and [23] solution for the XC7Z020 FPGA.

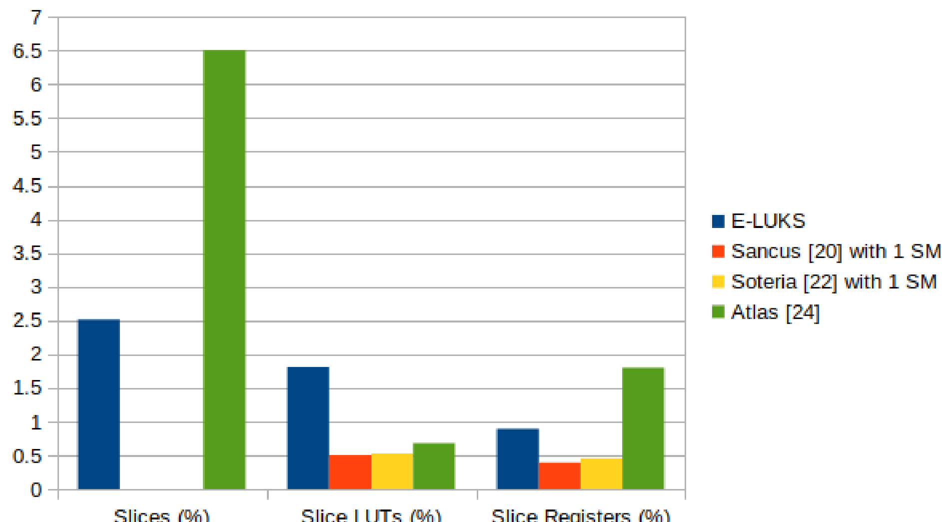


Figure 8. Percentage resources of E-LUKS and *Sancus*, *Soteria*, and *Atlas* solutions for the XC6VLX240T FPGA.

6. Conclusions

In this work, a hardware-solution able to perform FDE on resource constrained devices is presented. In addition, E-LUKS can authenticate and verify the integrity of the user data. The results presented show that E-LUKS takes few resources from the FPGA; less than 5%. However, there are other hardware-solutions, which provide security of the user data. There are the LUKS solutions, which, due to its pipeline implementation and the use of cryptographic algorithms, occupies ten times more resources, in the best of the scenarios, than E-LUKS. The Sancus and Soteria solutions have a smaller footprint than E-LUKS when working with few files. However, to handle a large amount of data, FDE could work better. These solutions need to implement specific instructions to the processor. E-LUKS, on the other hand, is processor agnostic, depending only on the bus system in order to create a bridge module. Atlas solution and E-LUKS have similar resource results. However, while Atlas only provides confidentiality, the work presented in [23], provides confidentiality, authentication, and integrity. Nonetheless, when providing, all three of them have a cost in terms of resources that is worse than E-LUKS, especially in Slice LUTs where it is almost three times greater. Additionally, in this scenario, Reference [23] needs part of the memory to store the TEC tree, which allows the authentication and integrity of the data. Hence, we believe that E-LUKS is an excellent choice for providing security to IoT devices. In those cases, the resources of the devices are the priority.

Our future work with E-LUKS involves Secure Boot, which we believe could be a perfect fit for this solution. Due to its agnostic processor characteristic and lack of software, it could be an excellent Secure Boot to IoT devices that take few resources and boot an embedded Linux image.

Author Contributions: Conceptualization, G.C.-Q., P.R.-d.-c.-V. and M.J.B.; methodology, G.C.-Q., P.R.-d.-c.-V. and M.J.B.; software, G.C.-Q. and P.R.-d.-c.-V.; validation, G.C.-Q., P.R.-d.-c.-V., M.J.B., D.G.-M., J.V.-C., J.J.-C. and E.O.-A.; formal analysis, G.C.-Q. and P.R.-d.-c.-V.; investigation, G.C.-Q., M.J.B. and P.R.-d.-c.-V.; resources, J.V.-C., J.J.-C., D.G.-M. and E.O.-A.; data curation, G.C.-Q. and P.R.-d.-c.-V.; writing—original draft preparation, G.C.-Q., P.R.-d.-c.-V. and M.J.B.; writing—review and editing, G.C.-Q., J.V.-C., J.J.-C., M.J.B., P.R.-d.-c.-V., E.O.-A. and D.G.-M.; visualization, G.C.-Q., P.R.-d.-c.-V. and M.J.B.; supervision, P.R.-d.-c.-V. and M.J.B.; project administration, J.J.-C. and P.R.-d.-c.-V.; and funding acquisition, J.J.-C. and P.R.-d.-c.-V. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by the Ministerio de Industria y Competitividad of Spain under project TIN2017-89951-P (BootTimeIoT) and by the European Regional Development Fund (ERDF).The author G.C.-Q. is economically supported by the VI-PPITUS.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Global Smart Shade Devices Market 2020–2024 | Emergence of IOT and AI-Based Smart Shade Devices to Boost Market Growth | Technavio | Business Wire. Available online: https://www.businesswire.com/news/home/20200131005322/en/Global-Smart-Shade-Devices-Market-2020-2024-Emergence (accessed on 14 May 2021).
2. Roaming Data Traffic Generated by Consumer & IoT Devices Forecast to Reach 2000 Petabytes in 2024: Kaleido Intelligence | Business Wire. Available online: https://www.businesswire.com/news/home/20200203005036/en/Roaming-Data-Traffic-Generated-Consumer-IoT-Devices (accessed on 14 May 2021).
3. Condry, M.W.; Nelson, C.B. Using Smart Edge IoT Devices for Safer, Rapid Response with Industry IoT Control Operations. *Proc. IEEE* **2016**, *104*, 938–946. [CrossRef]
4. IoT Medical Devices Market Growth Holds Strong; Key Players studied Medtronic, GE Healthcare, Philips Healthcare (Philips), Siemens—MarketWatch. Available online: https://www.marketwatch.com/press-release/iot-medical-devices-market-growth-holds-strong-key-players-studied-medtronic-ge-healthcare-philips-healthcare-philips-siemens-2020-01-30 (accessed on 20 May 2021).
5. Ferrer-Cid, P.; Barcelo-Ordinas, J.M.; Garcia-Vidal, J.; Ripoll, A.; Viana, M. Multi-sensor data fusion calibration in IoT air pollution platforms. *IEEE Internet Things J.* **2020**, *7*, 3124–3132. [CrossRef]
6. Gutierrez-Madronal, L.; La Blunda, L.; Wagner, M.F.; Medina-Bulo, I. Test Event Generation for a Fall-Detection IoT System. *IEEE Internet Things J.* **2019**, *6*, 6642–6651. [CrossRef]
7. Hamdan, O.; Shanableh, H.; Zaki, I.; Al-Ali, A.R.; Shanableh, T. IoT-Based Interactive Dual Mode Smart Home Automation. In Proceedings of the 2019 IEEE International Conference on Consumer Electronics, Las Vegas, NV, USA, 11–13 January 2019; pp. 1–2. [CrossRef]
8. Wazid, M.; Das, A.K.; Bhat K, V.; Vasilakos, A.V. LAM-CIoT: Lightweight authentication mechanism in cloud-based IoT environment. *J. Netw. Comput. Appl.* **2020**, *150*, 102496. [CrossRef]
9. Bodei, C.; Chessa, S.; Galletta, L. Measuring security in IoT communications. *Theor. Comput. Sci.* **2019**, *764*, 100–124. [CrossRef]
10. Meneghello, F.; Calore, M.; Zucchetto, D.; Polese, M.; Zanella, A. IoT: Internet of Threats? A Survey of Practical Security Vulnerabilities in Real IoT Devices. *IEEE Internet Things J.* **2019**, *6*, 8182–8201. [CrossRef]
11. binti Mohamad Noor, M.; Hassan, W.H. Current research on Internet of Things (IoT) security: A survey. *Comput. Netw.* **2019**, *148*, 283–294. [CrossRef]
12. Neshenko, N.; Bou-Harb, E.; Crichigno, J.; Kaddoum, G.; Ghani, N. Demystifying IoT Security: An Exhaustive Survey on IoT Vulnerabilities and a First Empirical Look on Internet-Scale IoT Exploitations. *IEEE Commun. Surv. Tutor.* **2019**, *21*, 2702–2733. [CrossRef]
13. Frustaci, M.; Pace, P.; Aloi, G.; Fortino, G. Evaluating critical security issues of the IoT world: Present and future challenges. *IEEE Internet Things J.* **2018**, *5*, 2483–2495. [CrossRef]
14. Fruhwirth, C.; Broz, M. LUKS1 On-Disk Format Specification. 2018. Available online: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/LUKS-standard/on-disk-format.pdf (accessed on 10 March 2021).
15. Shwartz, O.; Mathov, Y.; Bohadana, M.; Elovici, Y.; Oren, Y. Reverse Engineering IoT Devices: Effective Techniques and Methods. *IEEE Internet Things J.* **2018**, *5*, 4965–4976. [CrossRef]
16. Ling, Z.; Luo, J.; Xu, Y.; Gao, C.; Wu, K.; Fu, X. Security Vulnerabilities of Internet of Things: A Case Study of the Smart Plug System. *IEEE Internet Things J.* **2017**, *4*, 1899–1909. [CrossRef]
17. Manos, A.; Tim, A.; Michael, B.; Matt, B. Understanding the mirai botnet. In Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada, 16–18 August 2017; pp. 1093–1110.
18. Li, X.; Cao, C.; Li, P.; Shen, S.; Chen, Y.; Li, L. Energy-efficient hardware implementation of LUKS PBKDF2 with AES on FPGA. In Proceedings of the 2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, 23–26 August 2016; pp. 402–409. [CrossRef]
19. Li, X.; Wu, K.; Zhang, Q.; Lin, S.; Chen, Y.; Wong, S.Y. A High Throughput and Pipelined Implementation of the LUKS on FPGA. *J. Circuits Syst. Comput.* **2020**, *29*, 2050075. [CrossRef]
20. Noorman, J.; Agten, P.; Daniels, W.; Strackx, R.; Van Herrewege, A.; Huygens, C.; Preneel, B.; Verbauwhede, I.; Piessens, F. Sancus: Low-Cost Trustworthy Extensible Networked Devices with a Zero-Software Trusted Computing Base. In Proceedings of the 22nd USENIX Conference on Security, Washington, DC, USA, 14–16 August 2013; pp. 479–494.
21. Strackx, R.; Piessens, F.; Preneel, B. Efficient Isolation of Trusted Subsystems in Embedded Systems. In Proceedings of the Security and Privacy in Communication Networks, Singapore, 7–9 September 2010; Volume 50, pp. 344–361. [CrossRef]
22. Götzfried, J.; Müller, T.; De Clercq, R.; Maene, P.; Freiling, F.; Verbauwhede, I. Soteria: Offline software protection within low-cost embedded devices. *ACM Int. Conf. Proc.Ser.* **2015**, *7*, 241–250. [CrossRef]

23. Werner, M.; Unterluggauer, T.; Schilling, R.; Schaffenrath, D.; Mangard, S. Transparent memory encryption and authentication. In Proceedings of the 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 4–8 September 2017; pp. 1–6. [CrossRef]
24. Maene, P.; Götzfried, J.; Müller, T.; de Clercq, R.; Freiling, F.; Verbauwhede, I. Atlas: Application Confidentiality in Compromised Embedded Systems. *IEEE Trans. Dependable Secur. Comput.* **2019**, *16*, 415–423. [CrossRef]
25. Brož, M. *LUKS2 On-Disk Format Specification Version 1.0.0 Document History*; LUKS: Hong Kong, China, 2018; pp. 1–16.
26. rfc8018. *PKCS #5: Password-Based Cryptography Standard v2.1*; RSA Laboratories: Hebron, CT, USA, 2017.
27. Dworkin, M.; Barker, E.; Nechvatal, J.; Foti, J.; Bassham, L.; Roback, E.; Dray, J. *Advanced Encryption Standard (AES)*; NIST: Gaithersburg, MD, USA, 2001. [CrossRef]
28. Schneier, B.; Kelsey, J.; Whiting, D.; Wagner, D.; Hall, C. Twofish: A 128-Bit Block Cipher. *NIST AES Propos.* **1998**, *15*, 1–27. [CrossRef]
29. Anderson, R.; Biham, E.; Knudsen, L. Serpent: A proposal for the advanced encryption standard. *NIST AES Propos.* **1998**, *174*, 1–23.
30. Adams, D.C. *The CAST-128 Encryption Algorithm*; RFC 2144; Network Working Group. 1997. Available online: https://www.rfc-editor.org/rfc/rfc2144.txt (accessed on 15 April 2021).
31. Adams, D.C.; Gilchrist, J. *The CAST-256 Encryption Algorithm*; RFC 2612; Network Working Group. 1999. Available online: https://www.rfc-editor.org/rfc/rfc2612.txt (accessed on 15 April 2021).
32. Eastlake, D., 3rd; Jones, P. *US Secure Hash Algorithm 1 (SHA1)*; RFC 3174; Network Working Group. 2001. Available online: https://www.rfc-editor.org/rfc/rfc3174.txt (accessed on 15 April 2021).
33. Dang, Q. *Secure Hash Standard (SHS)*; NIST: Gaithersburg, MD, USA, 2012. [CrossRef]
34. Dobbertin, H.; Bosselaers, A.; Preneel, B. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption*; Gollmann, D., Ed.; Springer: Berlin/Heidelberg, Germany, 1996; pp. 71–82.
35. Pandey, J.G.; Goel, T.; Karmakar, A. Hardware architectures for PRESENT block cipher and their FPGA implementations. *IET Circuits Devices Syst.* **2019**, *13*, 958–969. [CrossRef]
36. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M.J.B.; Seurin, Y.; Vikkelsoe, C. PRESENT: An Ultra-Lightweight Block Cipher. In Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2007, Vienna, Austria, 10–13 September 2007; Paillier, P., Verbauwhede, I., Eds.; Springer: Berlin/Heidelberg, Germany, 2007; pp. 450–466.
37. Bogdanov, A.; Knežević, M.; Leander, G.; Toz, D.; Varıcı, K.; Verbauwhede, I. SPONGENT: A Lightweight Hash Function. In Proceedings of the Cryptographic Hardware and Embedded Systems—CHES 2011, Nara, Japan, 28 September–1 October 2011; Preneel, B., Takagi, T., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 312–325.
38. Bellare, M.; Canetti, R.; Krawczyk, H. Keying Hash Functions for Message Authentication. In Proceedings of the Advances in Cryptology — CRYPTO '96, Santa Barbara, CA, USA, 18–22 August 1996; Koblitz, N., Ed.; Springer: Berlin/Heidelberg, Germany, 1996; pp. 1–15.
39. Yao, F.F.; Yin, Y.L. Design and Analysis of Password-Based Key Derivation Functions. In Proceedings of the Topics in Cryptology—CT-RSA 2005, San Francisco, CA, USA, 14–18 February 2005; Menezes, A., Ed.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 245–261.
40. Xilinx. *7 Series FPGAs Data Sheet: Overview (DS180)*; Xilinx Inc.: San Jose, CA, USA, 2010.
41. Xilinx Inc. *Vivado Design Suite User Guide: UG973*; Xilinx Inc.: San Jose, CA, USA, 2020.
42. GitHub—Germancq/ELUKS. Available online: https://github.com/germancq/ELUKS (accessed on 25 October 2021).
43. GitHub—olofk/Fusesoc: Package Manager and Build Abstraction Tool for FPGA/ASIC Development. Available online: https://github.com/olofk/fusesoc (accessed on 8 October 2021).
44. Xilinx Inc. *Zynq-7000 AP SoC Technical Reference Manual*; Xilinx Inc.: San Jose, CA, USA, 2021.
45. Xilinx Inc. *Virtex-6 Family Overview Summary of Virtex-6 FPGA Features*; Xilinx Inc.: San Jose, CA, USA, 2015.

5.3. Address-encoder byte order

5.3.1. Breve resumen

Esta publicación presenta una modificación sobre el procesador OpenRISC. Dicha modificación habilita la posibilidad de interpretar por parte del procesador direcciones en Little-Endian como en Big-Endian indistintamente utilizando como base para este diseño los bits menos significativos del direccionamiento.

5.3.2. Datos Revista

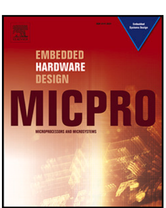
- Nombre Revista: Microprocessors and Microsystems
- Índice JCR(2020): 1.525(Q3)
- Fecha publicación: 12-09-2020
- DOI: `https://doi.org/10.1016/j.micpro.2020.103268`

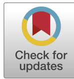
Microprocessors and Microsystems 78 (2020) 103268

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro





Address-encoded byte order

David Guerrero *, German Cano-Quiveu, Jorge Juan-Chico, Alejandro Millan, Manuel J. Bellido, Julian Viejo, Paulino Ruiz-de-Clavijo, Enrique Ostua

Departamento de Tecnología Electrónica, Universidad de Sevilla, 41012 Sevilla, Spain

ARTICLE INFO

Keywords:
ISA
Data alignment
Byte order
Endianness
Shared memory
MPSoC

ABSTRACT

Unaligned accesses are forbidden in many high-performance architectures. In most of these architectures, the least significant address bits of a multibyte memory access must be zero. Otherwise, the program generating the access is considered erroneous and an exception is flagged. The objective of this paper is to propose an alternative behaviour using the least significant address bits to encode the byte order of the accessed data. Modifying a traditional architecture to support the proposed behaviour presents several advantages, including backward compatibility at binary-code level, and the possibility of carrying out an endianness conversion during multibyte memory accesses without increasing the execution time nor using additional opcodes. The technique is demonstrated by modifying an OpenRISC 1000 implementation without introducing any penalty in hardware resources or performance. Subroutines written and compiled for the traditional architecture and originally designed for only the native byte order can, in the modified architecture, read and write data in a non-native byte order without any need to recompile. The execution of a sample algorithm operating on non-native byte order shows a reduction of 60% in the user execution time in the modified implementation when compared to the original implementation.

1. Introduction

In the memory model of most modern processors, memory locations are 8 bits wide, and therefore the minimum unit the processor can write or read from memory is a byte. The greatest number that can be represented by a byte using positional notation is $2^8 - 1 = 255$. To represent a greater integer, a processor must use a concatenation of bytes, known as *a word*. Using a single instruction, many processors can read or write a word greater than a byte, although the size of that word in bytes must usually be power of 2. Typical sizes for access are 2, 4, and 8 bytes, but many vector processors support larger sizes [1]. The designer of an architecture must decide two main issues when the size of a memory access can be greater than the size of the memory locations. One is *data alignment*, which determines the addresses permitted for accesses that are wider than a memory location. The other is *endianness*, which maps the memory locations involved in each access to chunks of the same size of the accessed word. Since this size is usually a byte, endianness is also referred to as *byte order*. For example, suppose that the word includes at least two bytes, Bm and Bl, and that the bits of Bm are more significant than the bits of Bl: In the *little-endian* byte order, Bm will be stored in a higher memory location than Bl, while in the *big-endian* byte order, Bm will be stored in a lower memory location than Bl [2,3]. Most architectures use one of the previous orders. For example, a picoJava processor uses the little-endian byte order [4], while the SPARC V8 architecture uses the big-endian byte order [5]. Several architectures, called bi-endian [6], use both. However, a few use other byte orders called *mixed-endian* (or *middle-endian*). In a mixed-endian order, the most significant half of a word is stored immediately after or immediately before the least significant half, depending on the size of that word. For example, in the vintage PDP-11 processor, words of two bytes are stored in little-endian order, that is, the most significant byte is stored after the least significant byte. If this processor stores a word of four bytes, then the word is split into two subwords of two bytes and the most significant subword is stored before the least significant subword, whereby each two-byte subword is still stored in little-endian order.

The communication between systems that use different byte orders is problematic [7]. Nowadays, heterogeneous Multiprocessor System-on-Chip (MPSoC) can include several processors with different endianness and therefore these problems can arise even when the systems are on the same chip [8–10]. For example, if a processor has to submit an integer to another processor with a different byte order through shared memory, then a protocol must be established. The first processor

* Corresponding author.
E-mail addresses: guerre@dte.us.es (D. Guerrero), germancq@dte.us.es (G. Cano-Quiveu), jjchico@dte.us.es (J. Juan-Chico), amillan@us.es (A. Millan), bellido@dte.us.es (M.J. Bellido), julian@us.es (J. Viejo), pruiz@us.es (P. Ruiz-de-Clavijo), ostua@dte.us.es (E. Ostua).

https://doi.org/10.1016/j.micpro.2020.103268
Received 2 December 2019; Accepted 10 September 2020
Available online 12 September 2020
0141-9331/© 2020 Elsevier B.V. All rights reserved.

could permute the bytes used to code the integer before writing them in accordance with the order used by the second processor. Alternatively, the first processor could write the bytes using its native order and the second should carry out the permutation after reading said bytes. Similar problems arise when dealing with files. Not all the file formats use the same byte order. If a system must read or write files with a format that uses a different byte order, then part of the executed instructions must be employed to reorder bytes [11]. Byte reordering also implies a severe penalty in emulators if the endianness of the guest and the host systems differ [12,13]. This penalty can be considerably reduced in architectures that have instructions to reorder the bytes of a register, such as the Intel 486 [14], although the penalty is not completely removed. It may be deemed that this presents no problem in bi-endian architectures, but it should be taken into account that, in certain architectures, application processes cannot change the endianness. This means that application processes must call the operating system to use the alien byte order. Worse still, library functions will probably be designed to use the native byte order and hence application processes will have to carry out a system call to switch to the native endianness before calling each function and then carry out another system call after every function call to return to the alien endianness. Similarly, if a function is going to use a byte order that differs from that used in the main code, then a system call must be carried out at the beginning of the function and another call before returning. Of course, each system call implies a severe overhead. In other bi-endian architectures, application processes can change the bit of the configuration register employed to set the endianness and hence they can switch the byte order with very little penalty [15]. In certain processors, the operation code specifies the endianness to be used and hence the penalty can be completely removed. For example, the native byte order of an x86 processor is little-endian, but if its Instruction Set Architecture (ISA) includes the MOVBE instruction, then it can carry out big-endian memory accesses [13]. Obviously, this instruction has its own operation code. Analogously, including instructions to support other mixed-endian byte orders would require additional operation codes. The purpose of this paper is to introduce an efficient way to encode and implement multi-endian support in architectures with data alignment restrictions.

The rest of the paper is organized as follows. In the next section, byte order and data alignment are formally defined. In Section 3, the proposed functionality is described. In Section 4, implementation details are discussed. Section 5 details how to write software to take advantage of the introduced functionality. Experimental performance results are shown in Section 6. The last section presents a summary of the conclusions.

2. Background

Hereinafter, we will use the following notation to describe a memory access:

- W: Word to be read or written.
- N: Size of W in bytes. In this paper we will assume this to be a power of 2.
- t: Logarithm of N to base 2. Hence $N = 2^t$.
- x: An element of $\mathbb{Z}_N = \{0, 1, 2, \ldots, N-1\}$.
- x_i: ith bit of the binary (base 2) representation of x. Hence $x = \sum_{i=0}^{t-1} x_i 2^i$.
- W_i: ith bit of W, whereby the concatenation of the bits W_{8N-1}, $\ldots$, W_1 and W_0 is W.
- B_x: concatenation of the bits W_{8x+7}, $\ldots$, W_{8x+1} and W_{8x+0}, hence the concatenation of the bytes B_{N-1}, $\ldots$, B_1 and B_0 is W.
- A: Address of the lowest memory location employed to store W.
- A_i: ith bit of the binary (base 2) representation of A.

Additionally, we will use the prefix $ for hexadecimal numerals. In the following subsections, we will detail decisions that must be made when designing an architecture whenever the size of the memory locations and the size of the words read or written may differ.

Table 1
Permutation used by the PDP-11 in four byte accesses.

x	0	1	2	3
$P_{\text{PDP4B}}(x)$	2	3	0	1

Table 2
Memory dumps of systems using different processors.

Address	$A+0$	$A+1$	$A+2$	$A+3$
Data picoJava	$40	$30	$20	$10
Data SPARC V8	$10	$20	$30	$40
Data PDP-11	$20	$10	$40	$30

2.1. Byte order

In most architectures, to access a word W of length N, the address A of the lowest memory location to be read or written must be provided. For example, if it is a write access, the component bytes $B_0, B_1, \ldots, B_{N-1}$ of W will be stored in the memory locations $A + 0, A + 1, \ldots, A + N - 1$, but not necessarily in that order. The byte order convention of the architecture maps those bytes to the memory locations. More formally, each byte B_x will be read or written in the memory location $A + P_N(x)$, where P_N is a permutation of $\mathbb{Z}_N$ as determined by the architecture. The most widely used permutations include the following:

- Identity function on $\mathbb{Z}_N$: This permutation is defined by $id_{\mathbb{Z}_N}(x) = x$.
- $(N-1)$'s complement: This permutation is defined by $C_{N-1}(x) = N - 1 - x$.

Little-endian architectures use the first permutation, while big-endian architectures use the second. The mixed-endian permutation used by the PDP-11 in four-byte accesses, which we denote as P_{PDP4B}, is shown in Table 1. As an example, Table 2 shows the content of the accessed memory locations of computers using a PicoJava processor (little-endian), a SPARC V8 processor (big-endian), and a PDP-11 processor (mixed-endian) immediately after storing the word $10203040 at address A.

Although other mixed-endian orders are rarely used, we will define them formally in order to explain the contribution introduced in this paper. In order to specify a mixed-endian order, it is necessary to set, for every word size greater than a byte, whether the most significant half of the word must be stored before or after the least significant half. Thus, if the size of the word is $N = 2^t$ (with $t > 0$) it is necessary to ascertain:

1. whether the most significant half of each word of size 2^1 must be stored before or after the least significant half.
2. whether the most significant half of each word of size 2^2 must be stored before or after the least significant half.

⋮

t. whether the most significant half of each word of size 2^t must be stored before or after the least significant half.

If the same decision is made for every word size, then the resulting order will be little-endian or big-endian and therefore little-endian and big-endian are particular cases of mixed-endian orders. In general, since for every size there are two options, the permutation of a mixed-endian order can be defined by a string of t bits. Let $m_{t-1} \ldots m_1 m_0$ be that string, the meaning of each bit m_i can be chosen arbitrarily. For example, the following convention can be used:

- $m_i = 0$: the most significant half of each word of size 2^{i+1} is stored in higher memory locations than those of the least significant half (as in little-endian).

- $m_i = 1$: the most significant half of each word of size 2^{i+1} is stored in lower memory locations than those of than the least significant half (as in big-endian).

We denote the permutation specified by the string $m_{t-1} \dots m_1 m_0$ using this convention as $Pl(m_{t-1}..m_1 m_0)$, since if every $m_i = 0$, then the little-endian order is set. This permutation can be computed with the formula

$$Pl(m_{t-1} \dots m_1 m_0)(x) = \sum_{i=0}^{t-1} (m_i \oplus x_i) 2^i \tag{1}$$

where $\oplus$ is the XOR operator and x_i is the bit at position i of the binary representation of x. This is simply a formal way to state that the binary representation of $Pl(m_{t-1} \dots m_1 m_0)(x)$ is a bitwise XOR of $m_{t-1} \dots m_1 m_0$ and $x_{t-1} \dots x_1 x_0$. The reason is simple: if $m_i = 0$ then the storing order of the two halves of each word of size 2^{i+1} cannot be different from the little-endian order and therefore the ith bits of the binary representations of x and $Pl(m_{t-1} \dots m_1 m_0)(x)$ must be equal; otherwise they must be different. An alternative convention can assign the opposite meaning to each bit m_i of the string $m_{t-1} \dots m_1 m_0$, that is:

- $m_i = 1$: the most significant half of each word of size 2^{i+1} is stored in higher memory locations than those of the least significant half (as in little-endian).
- $m_i = 0$: the most significant half of each word of size 2^{i+1} is stored in lower memory locations than those of the least significant half (as in big-endian).

We denote the permutation specified by the string $m_{t-1} \dots m_1 m_0$ using the second convention as $Pb(m_{t-1} \dots m_1 m_0)$, since if every $m_i = 0$, then the big-endian order is set. This permutation can be computed with the formula

$$Pb(m_{t-1} \dots m_1 m_0)(x) = \sum_{i=0}^{t-1} (m_i \odot x_i) 2^i \tag{2}$$

where $\odot$ is the XNOR operator. Again, this is a formal way to state that the binary representation of $Pb(m_{t-1} \dots m_1 m_0)(x)$ is a bitwise XNOR of $m_{t-1} \dots m_1 m_0$ and $x_{t-1} \dots x_1 x_0$. For example, the permutation used by the PDP-11 in four-byte accesses can be described by the string $m_1 m_0 = 10$ using the first convention, or by the string $m_1 m_0 = 01$ using the second convention. Note that $P_{\text{PDP4B}} = Pl(10) = Pb(01)$ as shown in Table 1.

2.2. Data alignment

By definition, a memory access to a word W of N bytes at address A is aligned if and only if A is multiple of N. The problems associated to unaligned accesses arise when designing the interconnection of the load/store units of a processor with the memory system. Such interconnection is exemplified in Fig. 1. The system in this example uses little-endian order, has addresses of P bits and general purpose registers of 2^r bytes with $r = 2$, although it can be easily extrapolated to other byte orders and values of r. When a word is read or written, the load/store unit provides to the memory system all the address bits except the r least significant, i.e. $A_{p-1} A_{p-2} \dots A_3 A_2$. The memory system can provide simultaneous access to the memory locations at the consecutive addresses $A_{p-1} \dots A_2 00$, $A_{p-1} \dots A_2 01$, $A_{p-1} \dots A_2 10$ and $A_{p-1} \dots A_2 11$. The load/store units generates the respective control signal EN_{00}, EN_{01}, EN_{10} and EN_1 to tell the memory system which of those four bytes memory locations will be accessed. Internally, the load/store units uses the r least significant address bits, i.e. A_1 and A_0, to map the accessed memory locations to the bytes of the word W to be read or written. For example, if a single byte at a logical address A such that $A_1 = 1$ and $A_0 = 0$ is accessed, then the data lines corresponding to the memory location $A_{p-1} \dots A_2 10$ will be connected to B_0, i.e. the bits $W_{7\dots0}$ of W, and only the control signal EN_{10} will be activated. The same happens if a word of two bytes at the same address is accessed, but additionally the data lines corresponding to the memory location

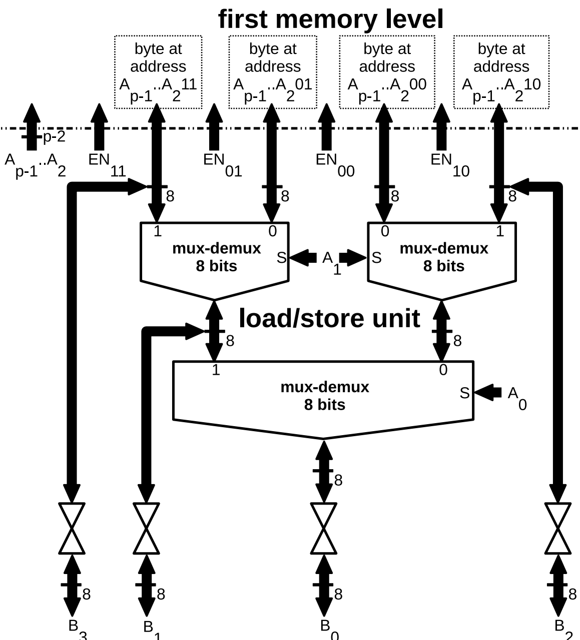


Fig. 1. Interconnection of a load/store unit with memory.

$A_{p-1} \dots A_2 11$ will be connected to B_1, and the control signal B_{11} will be also activated. In general, in an aligned access to a word of size $N = 2^t$, the logical address A of the lowest memory location of those employed to store the word is a multiple of 2^t, that is, the t least significant bits of this logical address are zero. Therefore, all the addresses in the range $[A, A + N - 1]$ match in the $P - t$ most significant bits. Since $t \le r$, this implies that the $P-r$ most significant bits of the logical addresses of the memory locations employed to store the word are the same and can be accessed simultaneously. From the load/store unit point of view, this involves a single access to a N-byte width memory.

In order to exemplify why unaligned accesses are more problematic, suppose that the load/store unit of Fig. 1 must read a word of two bytes at address 3. The bytes of this word are stored in memory locations 3 and 4. Since the bits A_2 of the addresses 3 and 4 fail to match, the load/store unit has to carry out two consecutive memory accesses to read the whole word. In general, any unaligned access has to be split into sub-accesses. Hence, implementing unaligned accesses involves several drawbacks:

- Unaligned accesses are slower.
- The second sub-access necessary to carry out an unaligned access can produce a page fault. Taking this into account complicates page fault management.
- The atomic read–modify–write operations are more complex since they may require two additional memory sub-accesses.
- The implementation of unaligned accesses requires hardware resources.

For these reasons, many high performance processors do not implement unaligned accesses. As previously mentioned, the t least significant address bits $A_{t-1} \dots A_1 A_0$ of any aligned access to a word of 2^t bytes are zero. Hereinafter, these bits are denoted *LSA* bits. If a processor does not support unaligned accesses, its behaviour when any of the LSA bits is not zero must be specified. Two alternative behaviours have been used in existing architectures:

1. One option, used in the Altivec ISA extension [16], is to simply ignore the t LSA bits. Therefore, when the processor is instructed to access a word at address A, the starting address of

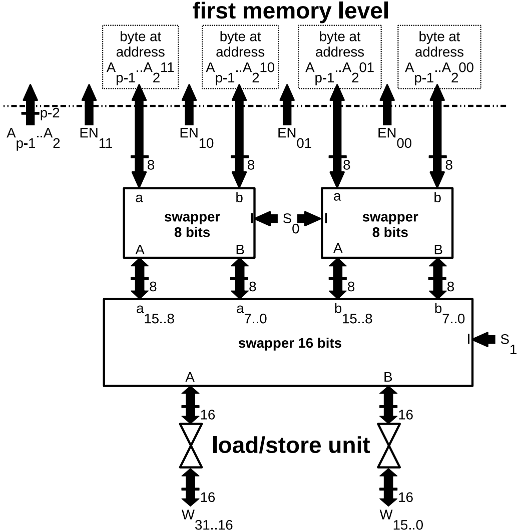


Fig. 2. Interconnection of a load/store unit implementing the proposed functionality with memory.

the word that will actually be accessed is $A - (A \bmod 2^t)$. An obvious advantage of this option is that it is easily implementable. Furthermore, no extra instructions are necessary to explicitly truncate (align-down) an address when handling addresses to the processor [17].

2. The most widely used option is to assume that a correct program will never produce such an access. The processor includes hardware to check that the t LSA bits are zero and, if any of these bits are one, then an exception is flagged to report that the program is erroneous.

In any of the previous options, when correct access to a word of 2^t bytes is carried out, the t LSA bits are meaningless and are not used. This paper introduces a third alternative behaviour by assigning semantics to those wasted bits that will be helpful when dealing with endianness conversion.

3. Semantics for the LSA bits

Our objective is to modify architectures that do not support unaligned accesses so that a LSA bit that is different from zero will no longer imply a programming error. Instead, it will indicate that the corresponding memory access must use a non-native byte order. We will define semantics for the LSA bits that will meet the following requirements:

1. If all the LSA bits are zero, then the memory access will use the byte order of the original architecture. This will ensure backward compatibility at binary-code level since no correct program written for the original architecture will produce a memory access with an LSA bit that differs from zero.
2. It must be possible to use several byte orders, including at least little-endian and big-endian, by choosing the values of the LSA bits of the memory access.
3. The modified architecture must be implementable without any time penalty with respect to the original architecture.

As in the Altivec ISA extension, when a word of size $N = 2^t$ is read or written using the effective address A, the starting address of the word will be $A - (A \bmod 2^t)$. However, in contrast to the Altivec ISA extension, the byte order will depend on the t LSA bits $A_{t-1} \ldots A_1 A_0$. In particular, the permutation $Pl(A_{t-1} \ldots A_1 A_0)$ will be used if the original architecture is little-endian, and the permutation $Pb(A_{t-1} \ldots A_1 A_0)$ will be used if the original architecture is big-endian, where Pl and Pb are the functions defined by the Eqs. (1) and (2). Note that if every LSA bit is zero (i.e. $A_{t-1} \ldots A_1 A_0 = 0 \ldots 00$), then $Pl(A_{t-1} \ldots A_1 A_0)(x) = x$ and $Pb(A_{t-1} \ldots A_1 A_0)(x) = N - 1 - x$, and therefore the permutation of the original architecture is used and the first requirement is met. Furthermore, if every LSA bit is one (i.e. $A_{t-1} \ldots A_1 A_0 = 1 \ldots 11$), then $Pl(A_{t-1} \ldots A_1 A_0)(x) = N - 1 - x$ and $Pb(A_{t-1} \ldots A_1 A_0)(x) = x$ and hence the permutation used corresponds to big-endian (if the original architecture is little-endian) or little-endian (if the original architecture is big-endian) and the second requirement is met. Moreover, it is possible to carry out a memory access using any mixed-endian permutation. For example, in order to read or write a word of four bytes using the mixed-endian permutation P_{PDP4B} of the PDP-11, we simply have to make the LSA bits $A_1 A_0$ equal to 10 if the native byte order is little-endian, or 01 if the native byte order is big-endian.

Since no correct program written for the original architecture will produce a memory access with an LSA bit that differs from zero, the unaligned access exception can be eliminated in the modified architecture without losing binary-code compatibility. However, for debugging purposes, it could be desirable to introduce a way to mask that exception instead of eliminating it. For example, many architectures include special purpose registers to store information regarding the state of execution of the current process. Several bits of these registers often have no associated meaning and are reserved for future versions of the architecture. The proposed variation of the architecture may use one of these bits to specify whether the unaligned access exception should be raised when the value of an LSA bit is not zero. If the operating system keeps a separate value of this special purpose register for every process, then every process can enable or disable the exception separately. If the special purpose register can only be modified in supervisor mode, then an user process that wants to use a non native byte order would have to call the operating system to disable the unaligned access exception. However, this implies very little overhead since the user process only has to make the call once. The process does not need to make any other system calls to disable the new functionality before calling library functions written for the old version of the architecture, since these functions will work as before.

4. Implementation

The implementation of the proposed variation of an architecture is straightforward. It simply requires a modification of the interconnection of the load/store units with memory and simple changes in the exception handling. As an example, Fig. 2 shows a modification of the interconnection circuit of Fig. 1 to implement the new functionality. In general, if the general purpose registers of the modified architecture have a size of 2^r bytes, then the memory system can provide simultaneous access to 2^r consecutive memory locations. The load/store circuitry includes r layers of swappers labelled from 0 to $r - 1$. Each swapper consists of two multiplexers/demultiplexers in parallel, as shown in Fig. 3. The set of swappers at level i will or will not swap both halves of each subword of size 2^{i+1} bytes depending on the value of the control signal S_i. If the native order of the architecture is little-endian, then each control signal S_i is connected directly to the address bit A_i. If the native order of the architecture is big-endian, then the value of each control signal S_i also depends on the size of the accessed word in the following way: if the accessed word has 2^t bytes then the value of each control signal S_i will be A_i if $i \geq t$ or $\overline{A_i}$ if $i < t$. Note that the new implementation introduces no time penalty since the delay of the interconnection circuits of Figs. 1 and 2 are the same, that is, twice the delay of a multiplexer/demultiplexer. Therefore, the third requirement described in the previous section is met.

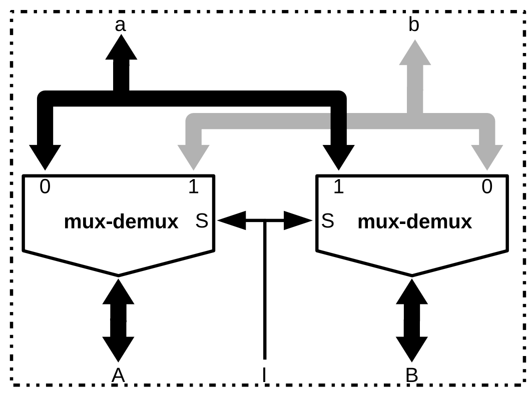


Fig. 3. Swapper circuit.

5. Software

In an architecture with the proposed functionality, each multibyte word W of size $N = 2^t$ stored in memory can be accessed using N different addresses that only differ in the t LSA bits. The lowest of these addresses must be used to read or write W using the native byte order of the architecture, and therefore we will call it *the native-endian address*. The highest of these addresses will be called *the reverse-endian address*. Hence, the address to be used for access in little-endian, that is, the little-endian address, is the native-endian address if the architecture is little-endian, but if the architecture is big-endian, then the little-endian address is the reverse-endian address. Analogously, the big-endian address is the native-endian address if the architecture is big-endian or the reverse-endian address if the architecture is little-endian. In general, any of the N mixed-endian byte orders can be used by selecting one of the N addresses of the word. It could be thought that taking advantage of this functionality is difficult since the programmer must use different addresses to access the same word depending on the byte order to be used. Fortunately, this can be easily solved in Reduced Instruction Set Computer (RISC) architectures by using assembler directives, since these architectures include the following features [18]:

- The only instructions with operands in memory are of load/store type.
- There are only a few data addressing modes, usually only immediate and register (which do not involve memory) and base plus displacement.

Therefore, to use a non-native byte order, it is only necessary to define a pseudoinstruction for every load/store instruction of the ISA. The assembler will replace every instance of the pseudoinstruction with the corresponding load/store instruction of the ISA with the same parameters, but a constant which depends on the desired byte order will be added to the offset. If this constant is simply the size of the accessed word in bytes minus one and it is added to a native-endian address, then the result will be the corresponding reverse-address. For example, suppose we have modified the OpenRISC 1000 32-bit big-endian architecture [19] with the proposed semantics. One of the instructions of its ISA is Load Half Word and Extend with Sign (`l.lhs`). We can use assembler directives to define an analogous little-endian pseudoinstruction `l.lelhs` so that every line in the form

```
l.lelhs rD,Ia(rA)
```

will be replaced with

```
l.lhs rD,Ib(rA)
```

where the offset `Ib` is simply `Ia` plus one (since the size of the access is two bytes). The same holds for store instructions, such as Store Single Word (`l.sw`): an analogous little-endian pseudoinstruction `l.lesw` can be defined so that every line in the form

```
l.lesw Ia(rA),rB
```

will be replaced with

```
l.sw Ib(rA),rB
```

where the offset `Ib` is `Ia` plus three (since the size of the access is four bytes). In this way, the endianness conversion is automatically carried out by the assembler. The programmer only needs to know that, as in the original architecture, every access must be aligned. Note that the memory access and the endianness conversion is carried out using a single assembler instruction.

Carrying out the endianness conversion in C is also very simple, even if the underlying architecture is not RISC, by using the macros of Fig. 4. The parameter of the `le` and `be` macros must be an l-value with a native endian address. The first parameter expands to an l-value with a little-endian address, while the second expands to an l-value with a big-endian address. They can be used to read variables in an alien byte order, but can also be used to write in an alien byte order when they are the left operand of an assignment. For example, suppose we want to assign to a variable `A` the value of another variable `B` multiplied by three. If both variables are stored using the native byte order, then the C code should be the following:

```
A=3*B;
```

If `B` is stored in little-endian and `A` must be stored in big-endian, then the code should be:

```
be(A)=3*le(B);
```

Note that, on the condition that the addresses of the arguments of the macros are known at compilation time, the corresponding binary code would have the same size and execution time and hence the endianness conversion introduces no penalty. To carry out endianness conversions involving arrays, the `lea` and `bea` macros can also be used. The parameter of these macros must be a pointer with a native-endian address to a word. The first macro expands to a pointer with little-endian address to the same word, while the second expands to a pointer with big-endian address to the same word. Again, when they are part of the left operand of an assignment, it is possible to write in an alien byte order. For example, suppose `A` is an one-dimensional array, `B` is a two-dimensional array, and the value `3*B[2][3]` must be assigned to `A[5]`. If both arrays are stored using the native byte order then the following code should be used:

```
A[5]=3*B[2][3];
```

If `B` is stored in little-endian and `A` must be stored in big-endian, then the code could be:

```
bea(A)[5]=3*lea(B)[2][3];
```

or alternatively:

```
be(A[5])=3*le(B[2][3]);
```

Again, the endianness conversion introduces no penalty. This can easily be extended to include any mixed-endian byte order. For example, the `pdp` and `pdpa` macros of Fig. 4 make it possible to use the byte order of the PDP-11 architecture.

More advantages arise when dealing with functions with parameters that are references. To exemplify this, suppose that a library has a function with the following interface:

```
#ifndef _AEBO_H
#define _AEBO_H
#ifndef _ENDIAN_H
#include <endian.h>
#endif
#if (__BYTE_ORDER != __LITTLE_ENDIAN && __BYTE_ORDER !=__BIG_ENDIAN)
#error "This version of the library does not support the host byte order yet."
#endif
#define __reverse_endian_address(x) ((typeof(x)) ((void *)(x) + sizeof(*(x)) - 1))
/*                                   ----------- ------------   -----------------
                                         (3)         (1)               (2)
   1. Convert to void* to have unit pointer arithmetic.
   2. Set less significant address bits to 1: use non-native (reverse) byte order.
   3. Restore the original pointer type.
*/
#define __reverse_endian(x) (*__reverse_endian_address(&(x)))
#if __BYTE_ORDER == __LITTLE_ENDIAN
#define le(x) (x)
#define lea(x) (x)
#define be(x) __reverse_endian(x)
#define bea(x) __reverse_endian_address(x)
#define pdpa(x) ((typeof(x)) ((void *)(x) + (sizeof(*(x))>2?2:0) ))
#define pdp(x) (*pdpa(&(x)))
#endif
#if __BYTE_ORDER == __BIG_ENDIAN
#define le(x) __reverse_endian(x)
#define lea(x) __reverse_endian_address(x)
#define be(x) (x)
#define bea(x) (x)
#define pdpa(x) ((typeof(x)) ((void *)(x) + (sizeof(*(x))>1?1:0) ))
#define pdp(x) (*pdpa(&(x)))
#endif
#endif
```

Fig. 4. Macros defined in aebo.h.

```
void foo(long* bar, int table[],...);
```

This is a ‘normal’ function in the sense that the parameters use the native byte order of the architecture. If we want the function to read and/or write the values of the arguments `native_bar` and `native_table` in the native endianness, it could be called in this way:

```
foo(&native_bar,native_table,...);
```

In a conventional architecture, if we wanted the function to be able to read and/or write one or more arguments using different byte orders, then it would be necessary to introduce at least one new parameter in its interface in order to select the desired endianness. Therefore, the new version of the function could be called in this way:

```
biendian_foo(endian_select,&bar,table,...);
```

This has several drawbacks:

- Obviously, the new function must be written and compiled.
- The new function cannot replace the old function because they have different interfaces and hence both versions must be maintained.
- It is necessary to include code in the new function to evaluate the new parameter and to act thereon. This introduces a penalty even if the ISA includes instructions to read and write memory in an alien byte order.

In contrast, if the architecture were extended to support the proposed functionality, then there would be no need to rewrite nor even recompile the function to use selectable endianness as long as it always accessed each word as a whole. For example, the arguments `my_bar` and `my_table` would be accessed by the function in big-endian and little-endian respectively, simply by calling it in this way:

```
foo(bea(&my_bar),lea(my_table),...);
```

Once more, there would be no penalty in the execution time of the function for using an alien byte order.

```
fread(table,size,1,input_file);
offset=dc_offset(table,length);
add_dc(-offset,table,length);
fwrite(table, size, 1, output_file);
```

Fig. 5. Simplified software filter that process data in native byte order.

6. Results

In order to measure the impact of the introduced functionality, the proposed modification has been applied to the 32-bit OpenRISC 1000 architecture. This architecture was chosen for the following reasons:

- Although an OpenRISC 1000 implementation supporting unaligned accesses can be carried out, the architecture establishes that, by default, unaligned accesses are not allowed and should trigger an exception [19]. For this reason, the proposed modification can be applied.
- There are open implementations that can easily be modified and synthesized for the reconfigurable hardware available to the authors [20].
- There are unrestricted operating systems and tool-chains available for the architecture that will allow performance measurements [21,22].

The original and modified architectures have been implemented, and programs with identical functionality have been executed in both implementations. These programs are versions of a basic filter written in C for the original architecture, which have been improved to process data in big-endian and little-endian. The original filter simply removes the DC offset of a set of 32-bit samples written in the native byte order. A simplification of its code is shown in Fig. 5.

Table 3
Synthesis results.

Architecture	Clock frequency	LUTs	Registers
Original	Maximum (100 MHz)	10 619	7779
Modified	Maximum (100 MHz)	10 552	7778

Table 4
Execution time comparison.

	Original time (s) Mean (Std. Dev.)	Modified time (s) Mean (Std. Dev.)	Speed up
User	0.505 (0.023)	0.200 (0.012)	60.46%
System	0.344 (0.040)	0.345 (0.030)	−0.26%
Total	0.865 (0.043)	0.559 (0.032)	35.33%

```
fread(table,size,1,input_file);
# if __BYTE_ORDER == __BIG_ENDIAN
if(selected_endian==little)
    for(index=0;index<length;index++)
        table[index]=le32toh(table[index]);
# else//host byte order is __LITTLE_ENDIAN
if(selected_endian==big)
    for(index=0;index<length;index++)
        table[index]=be32toh(table[index]);
# endif
offset=dc_offset(table,length);
add_dc(-offset,table,length);
# if __BYTE_ORDER == __BIG_ENDIAN
if(selected_endian==little)
    for(index=0;index<length;index++)
        table[index]=htole32(table[index]);
# else//host byte order is __LITTLE_ENDIAN
if(selected_endian==big)
    for(index=0;index<length;index++)
        table[index]=htobe32(table[index]);
# endif
fwrite(table, size, 1, output_file);
```

Fig. 6. Simplified improved filter for the original architecture.

6.1. Synthesis

An open implementation of the original architecture called *mor1kx* [20] and a modification to support the proposed functionality have been synthesized for the Digilent Nexys A7 Board [23] featuring a Xilinx Artix XC7A100T-CSG324 FPGA chip [24]. The only part of the data unit modified to support the proposed functionality is the combinational logic of the load/store unit, and the only difference in the behaviour of the control unit is that the unaligned access exception is disabled and hence the number of clock cycles required to execute each instruction are the same in both implementations. The full projects, including the bit streams, can be downloaded from [25] and [26]. The default options of the Vivado tool are used for synthesis. Results are as shown in Table 3. Since both implementations reached the maximum operation frequency supported by the board, i.e. 100 MHz, the execution time of a program written for the original architecture is the same in both. The employed resources were slightly reduced in the modified version.

6.2. Software test bench

The performance measurement is carried out under the Linux kernel 4.4.0 [21]. Since the proposed semantics ensures backward compatibility at binary-code level, the implementation of the modified architecture successfully runs the operating system of the original architecture without modifications. Furthermore, the execution time of the software written for the original architecture turns out to be the same. The

```
fread(table,size,1,input_file);
offset=dc_offset(selected_endian==big?
    bea(table) : lea(table),length);
add_dc(-offset,selected_endian==big?
    bea(table) : lea(table),length);
fwrite(table, size, 1, output_file);
```

Fig. 7. Simplified improved filter for the modified architecture.

improved versions of the software filter used in the test have to deal with the fact that the data may be written in little-endian byte order. It must be noted that the little-endian byte order is optionally supported by the OpenRISC 1000 architecture. If an implementation supported said order, then the little-endian byte order would be activated by setting the Little Endian Enable (LEE) bit of the SR register. However, it would be helpless in this case since it would not be possible to selectively change the endianness used by each process, and mor1kx does not support this functionality. Hence, the improved filter for the original architecture will carry out the endianness conversion by software using the `endian` library of `glibc`. A modification of the `dc_offset` and `add_dc` library functions used in the program for the original architecture to deal with arguments in an alien byte order would be inefficient since the input data should be converted twice. Instead, the input data will be converted to the native byte order before processing, and output data will be converted to the target byte order as shown in Fig. 6. The improved filter for the modified architecture is much simpler, since only the invocation of the functions have been modified, as shown in Fig. 7. The full program can be downloaded from [27]. The programs were compiled using the 5.3.0 version of the `or1k-linux-musl-gcc` tool-chain [22]. The execution time of both programs with random files of 250 K bytes stored in RAM was measured 100 times. The measurements can be repeated by executing the `test_bench.sh` shell script available in [27]. The results are shown in Table 4. The total execution time was reduced by ca. 35% with the proposed functionality, while the user execution time was reduced by over 60%. This result was expected since the OpenRISC 1000 ISA does not include instructions to reorder the bytes of a register and therefore the program for the original architecture had to carry out each endianness conversion using several logic and shift instructions. In particular, for 4-byte words, the `endian` library uses the internal C function `__bswap32`, which uses many instructions as shown in Fig. 8. The overhead of these instructions was removed in the modified version and hence the execution time was remarkably reduced. The size of the executable was also reduced from 9428 bytes to 9112 bytes, that is, a reduction of 3.35%.

7. Discussion

The processor modification was very straightforward, required negligible engineering cost and does not impact the performance when executing legacy code. Because of that it may be desirable independently of the expected applications of the processor.

8. Conclusions

Semantics for the least significant address bits of multibyte accesses in architectures that only support aligned accesses have been proposed. They have the following advantages:

- Existing architectures can easily be modified to include the proposed semantics so that a single instruction can carry out a memory access and a byte order conversion.
- The modification does not require new instructions nor operation codes since the least significant address bits are employed to code the byte order to be used.

```
__bswap32:
.cfi_startproc
l.sw     -4(r1),r2
.cfi_offset 2, -4
l.addi   r2,r1,0
.cfi_def_cfa_register 2
l.addi r1,r1,-8
l.sw     -8(r2),r3
l.lwz    r3,-8(r2)
l.srli   r4,r3,24
l.lwz    r3,-8(r2)
l.srli   r3,r3,8
l.andi   r3,r3,65280
l.or     r4,r4,r3
l.lwz    r3,-8(r2)
l.slli   r3,r3,8
l.movhi  r5,hi(16711680)
l.and    r3,r3,r5
l.or     r4,r4,r3
l.lwz    r3,-8(r2)
l.slli   r3,r3,24
l.or     r3,r4,r3
l.ori    r11,r3,0
l.ori r1,r2,0
l.lwz    r2,-4(r1)
l.jr     r9
l.nop
.cfi_endproc
```

Fig. 8. Assembler code corresponding to the _bswap32 function.

- Any architecture modified to include the proposed semantics will be backwards compatible at binary-code level and hence their implementations will be able to run any software written for the original architecture.
- The modification does not introduce any penalty.
- Assemblers and C compilers of the original architecture can easily be employed to take advantage of the new functionality by defining a few simple preprocessor macros.
- With the new functionality, subroutines written for the original architecture will be able to process referenced data in an alien byte order without any penalty being incurred. There is no need to rewrite nor even recompile these subroutines to take advantage of this functionality.
- An implementation of a traditional architecture was modified to support the proposed functionality. The modification did not introduce any penalty in hardware resources nor in execution time.
- A test bench processing data in a non native byte order intensively was executed in the original implementation and in the modified implementation. There was a reduction of over 35% of the total execution time and over 60% of the user execution time in the modified implementation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been partially supported by the Ministerio de Economía, Industria y Competitividad of Spain under project TIN2017-89951-P (BootTimeIoT) and by the European Regional Development Fund (ERDF).

References

[1] K. Cooper, L. Torczon, Engineering a Compiler, Elsevier Science, 2011, URL https://books.google.es/books?id=_tgh4bgQ6PAC.
[2] D. Cohen, On Holy Wars and a Plea for Peace, Computer 14 (10) (1981) 48–54, http://dx.doi.org/10.1109/C-M.1981.220208.
[3] IEEE standard for information technology–portable operating system interface (POSIX(R)) base specifications, Issue 7, in: IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), 2018, pp. 1–3951, http://dx.doi.org/10.1109/IEEESTD.2018.8277153.
[4] J.M. O'Connor, M. Tremblay, PicoJava-I: the Java virtual machine in hardware, IEEE Micro 17 (2) (1997) 45–53, http://dx.doi.org/10.1109/40.592314.
[5] M. Garcia, E. Francesquini, R. Azevedo, S. Rigo, HybridVerifier: A cross-platform verification framework for instruction set simulators, IEEE Embedded Syst. Lett. 9 (2) (2017) 25–28, http://dx.doi.org/10.1109/LES.2016.2626980.
[6] I. Horton, Beginning C++, Apress, Berkeley, CA, 2014, pp. 1–22, http://dx.doi.org/10.1007/978-1-4842-0007-0_1.
[7] M. Arora, The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits, Springer New York, New York, NY, 2012, pp. 155–168, http://dx.doi.org/10.1007/978-1-4614-0397-5_7.
[8] H.E. Yantir, A. Yurdakul, An efficient Heterogeneous register file implementation for FPGAs, in: 2014 IEEE International Parallel Distributed Processing Symposium Workshops, 2014, pp. 293–298, http://dx.doi.org/10.1109/IPDPSW.2014.40.
[9] J.-J. Li, S.-C. Wang, P.-C. Hsu, P.-Y. Chen, J.K. Lee, A multi-core software API for Embedded MPSoC environments, in: Proceedings of the Second Russia-Taiwan Conference on Methods and Tools of Parallel Programming Multicomputers, in: MTPP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 40–50, URL http://dl.acm.org/citation.cfm?id=1927517.1927524.
[10] J. Henkel, S. Parameswaran (Eds.), Designing Embedded Processors: A Low Power Perspective, Springer Netherlands, 2007.
[11] M.A.A. Farhan, D.E. Keyes, Optimizations of unstructured aerodynamics computations for many-core architectures, IEEE Trans. Parallel Distrib. Syst. 29 (10) (2018) 2317–2332, http://dx.doi.org/10.1109/TPDS.2018.2826533.
[12] R. Auler, E. Borin, The case for flexible ISAs: Unleashing hardware and software, in: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2017, pp. 65–72, http://dx.doi.org/10.1109/SBAC-PAD.2017.16.
[13] G. Kondoh, H. Komatsu, Dynamic Binary translation specialized for embedded systems, SIGPLAN Not. 45 (7) (2010) 157–166, http://dx.doi.org/10.1145/1837854.1736019, URL http://doi.acm.org/10.1145/1837854.1736019.
[14] M. Souza, D. Nicácio, G. Araújo, ISAMAP: Instruction Mapping driven by dynamic binary translation, in: A.L. Varbanescu, A. Molnos, R. van Nieuwpoort (Eds.), Computer Architecture, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 117–138.
[15] A. Sloss, D. Symes, C. Wright, in: M. Kaufmann (Ed.), ARM System Developer's Guide, 2004, p. 689.
[16] K. Diefendorff, P.K. Dubey, R. Hochsprung, H. Scale, AltiVec extension to PowerPC accelerates media processing, IEEE Micro 20 (2) (2000) 85–95, http://dx.doi.org/10.1109/40.848475.
[17] J. Rentzsch, Data alignment: Straighten up and fly right, 2005, URL https://www.ibm.com/developerworks/library/pa-dalign/index.html.
[18] J.L. Hennessy, a.A. Patterson, Computer Architecture: A Quantitative Approach, Elsevier, 2011.
[19] OpenRISC 1000 Architecture Manual, 2012, URL http://opencores.org/websvn,filedetails?repname=openrisc&path=%2Fopenrisc%2Ftrunk%2Fdocs%2Fopenrisc-arch-1.0-rev0.pdf.
[20] mor1kx IP core specification, URL https://github.com/openrisc/mor1kx/blob/master/doc/mor1kx.asciidoc.
[21] Linux kernel 4.4.0 for OpenRISC, URL https://github.com/openrisc/linux/tree/for-next/kernel.
[22] or1k-linux-musl-gcc tool-chain, URL https://github.com/openrisc/musl-cross.
[23] Nexys A7 FPGA Trainer Board, URL https://store.digilentinc.com/nexys-a7-fpga-trainer-board-recommended-for-ece-curriculum/.
[24] Artix-7 devices, URL https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html.
[25] mor1kx synthesized for the Digilent Nexys 4 DDR board, URL https://gitlab.com/davidguerrero/mor1kx-synthesized-for-the-digilent-nexys-4-ddr.
[26] A mor1kx implementation modified to implement a variation of the 32 bit OpenRISC 1000 architecture with multiendian capabilities and synthesized for the Digilent Nexys 4 DDR board, URL https://gitlab.com/davidguerrero/mor1kx-multiendian.
[27] Software test bench to compare the 32 bit OpenRISC 1000 architecture with a variation with multiendian capabilities running Linux, URL https://gitlab.com/davidguerrero/openrisc-1000-multiendian-test-bench.

David Guerrero Martos received his Bsc degree and the Ph.D. degree in Computer Engineering from the University of Seville, Spain, in 2000 and 2012, respectively. Since 2002, he has been working as a lecturer in the Department of Electronics Technology of said university. His research interests include digital circuit synchronization, hardware implementation of numerical methods, and computer architecture. He has published several papers in journals and conferences.

German Cano received his Bsc degree in computing engineering from the University of Seville, Spain, in 2015. He has been a Ph.D. student in the Electronics Technology Department at this university since 2017. His research interests include Bootloaders, IoT, and SoC.

Dr. Jorge Juan received his Bsc degree (1994) and Ph.D. degree in Physics (2000) from the University of Seville, Spain. He is currently a lecturer in the Electronics Technology Department at this university where he is leading the Digital Research and Development Group. Dr. Juan has carried out research in the areas of metastability, delay modelling, timing and power simulation, and digital embedded systems.

Alejandro Millan was born in Seville in 1975. He received his M.Sc. in Computer Engineering in 1999 and his Ph.D. in 2008, from the University of Seville. He works as a Professor at its Department of Electronics Technology. Since 1999, he has taught at the School of Computer Engineering and the Polytechnic School of Engineering. He is a member of its ID2 Research Group where he has participated in 20 research projects, and he has published in excess of 50 conference papers and a total of 17 journal papers.

Dr. Manuel J. Bellido received his B.Sc. degree (1987) and Ph.D. degree (1994) in Physics from the University of Seville, Spain. He has been with the Electronics Technology Department at this university since 1990 where he holds a post as a Professor.

Julian Viejo received his M.Sc. and Ph.D. degrees in Computing Engineering from the University of Seville (Spain) in 2004, and 2011, respectively. He works as an assistant professor in the Department of Electronics Technology of that University and has contributed several research papers to international journals and conferences in the area of Digital Signal Processing and System-on-Chip design.

Paulino Ruiz-de-Clavijo received his Bsc degree (1999) and Ph.D. degree (2007) in computer science from the University of Seville (Spain). He has been with the Electronics Technology Department at this university since 1999 where he has held a post as an assistant professor since 1999. He was also with the Institute of Microelectronics of Seville, part of the National Centre of Microelectronics in Spain, from 1998 to 2004. His research work includes system-on-chip designs, Digital Signal Processing, and embedded microprocessors architecture: areas to which he has contributed in international conferences and workshops.

Enrique Ostua received his B.Sc. degree in computing engineering (2003) and his M.Sc. in computing & networking engineering (2015) from the University of Seville. He has been an associate professor in the Department of Electronics Technology, University of Seville, since 2003. His research interests include system-on-chip designs and embedded microprocessor architecture areas, to which he has contributed in international journals, conferences, and workshops.

5.4. OpenRISC hardware bootloader over WiFi

5.4.1. Breve resumen

En esta conferencia, se presenta una implementación sobre FPGA de un bootloader hardware, con la particularidad de ejecutar el arranque del sistema operativo Linux en remoto mediante Wi-Fi utilizando paquetes de datos de 512 Bytes. Para la comunicación inalámbrica se utilizo el modulo ESP-8266 y como placa de desarrollo la Nexys4DDR de Digilent.

5.4.2. Datos Conferencia

- Nombre Conferencia: The open source digital design conference (ORConf)
- Ubicación: Gdansk, Polonia
- Fecha presentación: 22-09-2018
- Formato presentación: Diapositivas
- Autores : Germán Cano Quiveu
- Menciones : Finalista del Xilinx Open Hardware Competition http://www.openhw.eu/2018
- Enlace : https://orconf.org/2018/#openriscwifi

Capítulo 6

Conclusiones y Trabajo futuro

En este último capítulo que compone esta tesis por compendio de artículos se establecerán las conclusiones extraídas así como el trabajo futuro derivados de los documentos aquí recogidos.

6.1. Conclusiones

Esta tesis, junto con los trabajos que la componen, ha sido concebida para formar parte de la linea «Bootloaders y Sistemas de archivos para Sistemas empotrados» del proyecto de investigación TIN2017-89951-P: *BootTimeIoT: Sistemas de inicio avanzados y sincronización temporal de alta precisión para IoT*. Esta línea del proyecto tenía como fin desarrollar un sistema de arranque (bootloader) con un uso de recursos hardware mínimo y aplicable a cualquier microprocesador específico de sistemas empotrados. Por tanto, respecto a este objetivo y con los resultados derivados de esta tesis podemos concluir que:

- Se ha desarrollado un Core Bootloader, escrito es su totalidad en SystemVerilog sin necesidad de IPCores propietarios, que consta de una interfaz SPI para obtener la imagen del sistema operativo y almacenarla en la memoria, tras lo cual, comandara al microprocesador para su ejecución.
- Se han implementado diferentes mecanismos para el almacenamiento de la imagen y su obtención, como son:
  - Carga desde una memoria flash externa, como es una tarjeta microSD.
  - Carga OTA (*Over The Air*), utilizando para ello una conexión WiFi.
- Las características del Core Bootloader permite que sea aplicable a cualquier combinación de microprocesador y bus. Además, el hecho de que este implementado sin utilizar IPCores propietarios permite su uso sobre cualquier FPGA con independencia de su fabricante.
- Se ha testeado el Core Bootloader con el correcto funcionamiento de carga de un kernel de Linux sobre OpenRISC junto con el bus Wishbone

[Wis], sobre la placa de desarrollo Nexys4DDR de Digilent con FPGA XC7A100T-1CSG324 de Xilinx [Dig].

Sin embargo, durante el desarrollo de este Core Bootloader emergió un problema importante a raíz de las modificaciones necesarias para su refinamiento, este proceso fue arduo, especialmente debido a la depuración que conllevaba. Es por ello, que se requirió de crear una metodología para tanto el desarrollo de Cores así como de su verificación. Esto derivo en una tarea fundamental para continuar la linea de trabajo y llevo a la consecución del articulo *An Integrated Digital System Design Framework With On-Chip Functional Verification and Performance Evaluation* cuyo resultados arrojaron las siguientes conclusiones:

- Se ha desarrollado una metodología de verificación integral capaz de agilizar el proceso de diseño e implementación de un Core.
- La metodología permite el uso de cualquier TPG y reutilizarlo tanto en la fase de simulación RTL como en la verificación On-Chip, este proceso de generación esta automatizado mediante scripts en Python.
- La verificación On-Chip muestra un aumento significativo en lo referente a velocidad de procesamiento de patrones comparado con otras herramientas generales del mercado como puede ser Xilinx VIO, este aumento es del orden de dos ordenes de magnitud superior, en el peor de los casos.
- El hecho de usar una memoria flash externa para almacenar los patrones de test y sus resultados, permite a la metodología verificar On-Chip el Core sin tener que recurrir a un computador externo.
- La metodología esta realizada sin utilizar IPCores propietarios siendo en su totalidad *Open Source*, para poder abarcar el mayor numero de plataformas posibles.

Por otra parte, el desarrollo del Core Bootloader trajo también otra cuestión sobre la mesa, la seguridad. El bootloader implementado no contemplaba el hecho de ningún ataque por entidades externas, como podría ser una imagen modificada en la microSD o un *man in the middle* en la transmisión OTA. Todo esto llevo a diseñar un Core que pudiera subsanar estos fallos de seguridad, siempre teniendo en cuenta la línea del proyecto donde se enmarca la tesis, por tanto, este Core debía ser diseñado específicamente para sistemas empotrados de bajos recursos, esto fue el centro del articulo *Embedded LUKS (E-LUKS): A Hardware Solution to IoT Security*, este trabajo presenta el Core E-LUKS, del cual podemos concluir:

- E-LUKS, esta basado en el sistema LUKS de Linux, modificando su estructura interna y algoritmos criptográficos para poder ser adecuado a dispositivos IoT de bajos recursos.
- E-LUKS confiere al sistema confidencialidad, integridad y autenticidad sin requerir por ello modificar el procesador/SoC donde se integre.

- Los resultados de E-LUKS en cuanto al uso de recursos consumidos del sistema, muestran que reduce el uso de recursos respecto a la mayoría de soluciones especificas para IoT de bajos recursos e iguala en la minoría restante. Respecto a LUKS, la inspiración de E-LUKS, muestra E-LUKS es muy superior en este aspecto.
- E-LUKS ha sido desarrollado bajo la filosofía *Open Source*, lo cual lleva a que pueda ser integrado en cualquier plataforma. Además, ha sido implementado de tal manera que su inclusión en cualquier sistema no interfiera, solo requiriendo un modulo puente para conectarse al bus del sistema.

Por último, se quiere enfatizar el hecho de que estos dos últimos trabajos han sido desarrollados como *Open Source*, por tanto todos los códigos así como los resultados discutidos aquí en la tesis se encuentran en los siguientes repositorios de github:

- Metodologia integral: `https://github.com/germancq/IPCores`.
- E-LUKS: `https://github.com/germancq/ELUKS`.

6.2. Trabajo Futuro

Concluyendo con las líneas de trabajo futuro que nos abre esta tesis, existen dos sobre las que se están trabajando actualmente. La primera sería crear un analizador sintáctico de SystemVerilog el cual pueda generar automáticamente la integración del Autotest Core con un Core cualquiera. La solución actual se basa en unas plantillas que requieren de una mínima preparación, especialmente indicar el número de señales y su longitud.

La segunda de las líneas, y la más prometedora, vendría dada por las posibilidades abiertas por el Core E-LUKS. Tal y como se ha especificado reiteradamente durante esta tesis, el proyecto de investigación TIN2017-89951-P nos conduce hacia sistemas de arranque y sistemas de ficheros para dispositivos IoT. Además, hay que incluir en este proyecto el sistema de ficheros NanoFS [Cla+13], con el cual se ha trabajado en estos derroteros. Con todo esto se esta trabajando en ciertos proyectos actualmente, tales como:

- Desarrollar un Core NanoFS para implementar el sistema de ficheros usando la mínima cantidad de recursos hardware. Pudiendo así ser utilizado en diversos sistemas.
- Crear un *Secure Boot* basado en E-LUKS para procesadores RISC.
- Crear un Core capaz de leer particiones NanoFS sobre E-LUKS sin ninguna necesidad de requerimiento software por parte del microprocesador.
- Desarrollar un Core que habilite un sistema de ficheros seguro usando como referencias E-LUKS y NanoFS.

Bibliografía: Artículos derivados de la tesis

[CQ+21a] G. Cano-Quiveu y col. «An Integrated Digital System Design Framework with On-Chip Functional Verification and Performance Evaluation». En: *IEEE Access* 9 (2021), págs. 161383-161394. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3132188.

[CQ+21b] G. Cano-Quiveu y col. «Embedded LUKS (E-LUKS): a hardware solution to IoT security». En: *Electronics* 10.23 (2021), págs. 1-22. ISSN: 2079-9292. DOI: 10.3390/electronics10233036.

[CQ18] G. Cano-Quiveu. «OpenRISC hardware bootloader over WiFi». En: *The open source digital design conference (ORConf)*. Gdansk (Poland), 2018.

[GM+20] D. Guerrero-Martos y col. «Address-encoded byte order». En: *Microprocessors and Microsystems (MicPro)* 78 (2020), págs. 1-9. ISSN: 0141-9331. DOI: 10.1016/j.micpro.2020.103268.

Bibliografía

[BCK96] Mihir Bellare, Ran Canetti y Hugo Krawczyk. «Keying Hash Functions for Message Authentication». En: *Advances in Cryptology — CRYPTO '96*. Ed. por Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, págs. 1-15. ISBN: 978-3-540-68697-2.

[Ber03] S. Bertman. *Handbook to Life in Ancient Mesopotamia*. Facts on File library of world history. Facts On File, Incorporated, 2003. ISBN: 9780816074815. URL: https://books.google.es/books?id=3bX3HYm5YMAC.

[Bog+07] Andrey Bogdanov y col. «PRESENT: An ultra-lightweight block cipher». En: *International workshop on cryptographic hardware and embedded systems*. Springer. 2007, págs. 450-466.

[Bog+11] Andrey Bogdanov y col. «spongent: A Lightweight Hash Function». En: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. por Bart Preneel y Tsuyoshi Takagi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, págs. 312-325. ISBN: 978-3-642-23951-9.

[Cla+13] P. Ruiz de Clavijo y col. «NanoFS: a hardware-oriented file system». En: *Electronics Letters* 49.19 (2013), págs. 1216-1218. ISSN: 0013-5194. DOI: 10.1049/el.2013.1961.

[Cla+17] P. Ruiz de Clavijo y col. «Minimalistic SDHC-SPI hardware reader module for boot loader applications». En: *Microelectronics Journal* 67 (2017), págs. 32-37. ISSN: 0026-2692. DOI: 10.1016/j.mejo.2017.07.007.

[Coc] *Introduction — cocotb 1.1 documentation*. URL: https://cocotb.readthedocs.io/en/latest/introduction.html (visitado 08-04-2019).

[Dig] *Digilent Nexys4 DDR Board*. URL: https://www.xilinx.com/support/university/boards-portfolio/xup-boards/DigilentNexys4DDR.html#overview (visitado 27-02-2022).

[DN+09] Giorgio Di Natale y col. «Self-test techniques for crypto-devices». En: *IEEE transactions on very large scale integration (VLSI) systems* 18.2 (2009), págs. 329-333.

[Env] *Cuneiform Tablets and 'Envelopes' Tell of Mesopotamian Sophistication | Ancient Origins*. URL: https://www.ancient-origins.net/news-history-archaeology/mesopotamian-cuneiform-tablets-0012947 (visitado 29-04-2021).

[Esp] *GitHub - wallento/wb2axi: Wishbone to ARM AMBA 4 AXI*. URL: https://github.com/wallento/wb2axi (visitado 15-02-2022).

[FB18] Clemens Fruhwirth y Milan Broz. «LUKS1 On-Disk Format Specification». En: (2018).

[Fos18] Harry D. Foster. «2018 FPGA Functional Verification Trends». En: *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. 2018, págs. 40-45. DOI: 10.1109/MTV.2018.00018.

[Fpg] *Field Programmable Gate Array Market Size Report, 2020-2027*. URL: https://www.grandviewresearch.com/industry-analysis/fpga-market (visitado 31-01-2022).

[Fus] *GitHub - olofk/fusesoc: Package manager and build abstraction tool for FPGA/ASIC development*. URL: https://github.com/olofk/fusesoc (visitado 03-03-2020).

[Göt+15] Johannes Götzfried y col. «Soteria: Offline software protection within low-cost embedded devices». En: *ACM International Conference Proceeding Series* 7-11-Decem (2015), págs. 241-250. DOI: 10.1145/2818000.2856129.

[Hai+19] Zeeshan Haider y col. «A Low-Cost Self-Test Architecture Integrated with PRESENT Cipher Core». En: *IEEE Access* 7 (2019), págs. 46045-46058. ISSN: 21693536. DOI: 10.1109/ACCESS.2019.2907717.

[Iot] *Industrial Internet Of Things Market Size Report, 2021-2028*. URL: https://www.grandviewresearch.com/industry-analysis/industrial-internet-of-things-iiot-market (visitado 31-01-2022).

[Kah96] D. Kahn. *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet*. Scribner, 1996. ISBN: 9781439103555. URL: https://books.google.es/books?id=3S8rhOEmDIIC.

[KR11] Lars R. Knudsen y Matthew J.B. Robshaw. *The Block Cipher Companion*. Information Security and Cryptography. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-17341-7. DOI: 10.1007/978-3-642-17342-4. URL: http://link.springer.com/10.1007/978-3-642-17342-4.

[Li+16] Xiaochao Li y col. «Energy-efficient hardware implementation of LUKS PBKDF2 with AES on FPGA». En: *Proceedings - 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, 10th IEEE International Conference on Big Data Science and Engineering and 14th IEEE International Symposium on Parallel and Distributed Proce* (2016), págs. 402-409. DOI: 10.1109/TrustCom.2016.0090.

[Li+20] Xiaochao Li y col. «A High Throughput and Pipelined Implementation of the LUKS on FPGA». En: *Journal of Circuits, Systems and Computers* 29.05 (2020), pág. 2050075. DOI: 10.1142/S0218126620500759. eprint: https://doi.org/10.1142/S0218126620500759. URL: https://doi.org/10.1142/S0218126620500759.

[Mae+19] Pieter Maene y col. «Atlas: Application Confidentiality in Compromised Embedded Systems». En: *IEEE Transactions on Dependable and Secure Computing* 16.3 (2019), págs. 415-423. DOI: 10.1109/TDSC.2018.2858257.

[Men+19] Francesca Meneghello y col. «IoT: Internet of Threats? A Survey of Practical Security Vulnerabilities in Real IoT Devices». En: *IEEE Internet of Things Journal* 6.5 (2019), págs. 8182-8201. ISSN: 23274662. DOI: 10.1109/JIOT.2019.2935189.

[MF21] Arno Mittelbach y Marc Fischlin. *The Theory of Hash Functions and Random Oracles*. 2021. ISBN: 978-3-030-63286-1. URL: http://link.springer.com/10.1007/978-3-030-63287-8.

[Mig] «Xilinx 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide for HDL». En: 768 (2013).

[ND12] Srinivas Nidhra y Jagruthi Dondeti. «Black box and white box testing techniques-a literature review». En: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), págs. 29-50.

[Noo+13] Job Noorman y col. «Sancus: Low-Cost Trustworthy Extensible Networked Devices with a Zero-Software Trusted Computing Base». En: *Proceedings of the 22nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013, 479–494. ISBN: 9781931971034.

[Ope] *OpenRISC - OpenRISC*. URL: https://openrisc.io/ (visitado 28-01-2022).

[PP10] Christof Paar y Jan Pelzl. *Understanding Cryptography*. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-04101-3.

[Sha49] C. E. Shannon. «Communication Theory of Secrecy Systems*». En: *Bell System Technical Journal* 28.4 (1949), págs. 656-715. DOI: https://doi.org/10.1002/j.1538-7305.1949.tb00928.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1949.tb00928.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1949.tb00928.x.

[Shi07] S Shioda. «Some upper and lower bounds on the coupon collector problem». En: *Journal of Computational and Applied Mathematics* 200.1 (2007), págs. 154-167.

[Shw+18] Omer Shwartz y col. «Reverse Engineering IoT Devices: Effective Techniques and Methods». En: *IEEE Internet of Things Journal* 5.6 (2018), págs. 4965-4976. ISSN: 23274662. DOI: 10.1109/JIOT.2018.2875240.

[Str] «An Overview of BIST». En: *A Designer's Guide to Built-In Self-Test*. Boston, MA: Springer US, 2002, págs. 1-14. ISBN: 978-0-306-47504-7. DOI: 10.1007/0-306-47504-9_1. URL: https://doi.org/10.1007/0-306-47504-9_1.

[Wb2] *GitHub - wallento/wb2axi: Wishbone to ARM AMBA 4 AXI*. URL: https://github.com/wallento/wb2axi (visitado 15-02-2022).

[Wer+17] Mario Werner y col. «Transparent memory encryption and authentication». En: *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 2017, págs. 1-6. DOI: 10.23919/FPL.2017.8056797.

[Wis] *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores — WISHBONE B3*. URL: https://wishbone-

interconnect.readthedocs.io/en/latest/index.html (visitado 27-02-2022).

[Xil10] Xilinx. «7 Series FPGAs Data Sheet: Overview (DS180)». En: 180 (2010), págs. 1-18. URL: www.xilinx.com.

[YY05] Frances F. Yao y Yiqun Lisa Yin. «Design and Analysis of Password-Based Key Derivation Functions». En: *Topics in Cryptology – CT-RSA 2005*. Ed. por Alfred Menezes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, págs. 245-261. ISBN: 978-3-540-30574-3.