PROYECTO FINAL DE INGENIERÍA

CRIPTOGRAFIA LIVIANA PARA OBJETOS CONECTADOS

BERAZA, LEANDRO MARTÍN – LU: 95816

Ingeniería en Informática

Tutor:
WEHBE, Ricardo Abraham - UADE

2021

UNIVERSIDAD ARGENTINA DE LA EMPRESA

FACULTAD DE INGENIERÍA Y CIENCIAS EXACTAS

Agradecimientos

A mi tutor Dr. Ricardo Wehbe, por haberme provisto de su invaluable conocimiento, por haberme guiado y encauzado con sus comentarios y apuntes.

A los directores Mg. Hernán Morello y Mg. Aníbal Freijo, por su destacada labor en UADE en lo que incumbe a esta carrera, por haber estado presente y contestado mis inquietudes.

A todos los profesores, y en especial a aquellos que me han brindado su apoyo y tiempo fuera del horario de clases.

A todos mis compañeros de equipo de los cuales he formado parte. He aprendido de todos ellos también.

A los próceres Alberto y Sergio, fuentes inagotables de conocimiento matemático.

A los impulsores de esta meta, directa o indirectamente: Fernando, Roberto, mi abuelo Ramón, y por sobre todo mi madre, gran ponderadora del arte del conocimiento.

Y a mi familia, por acompañarme en este viaje y ser el insoslayable soporte emocional.

A todos ellos: ¡Muchas gracias!

Resumen

Los sistemas conectados son dispositivos de pocos recursos que necesitan ser protegidos con una criptografía diseñada contemplando sus carencias en dichos recursos. Esa criptografía se denomina "criptografía liviana" y este Proyecto Final de Ingeniería aborda la rama dedicada a las funciones hash. La misma presenta una serie de características, como son: procesar un mensaje de longitud arbitraria y obtener una salida pseudoaleatoria de longitud fija; esa función debe ser fácil de calcular, pero difícil de recorrer el camino inverso. Dado que la longitud de la salida es finita, el proceso es propenso a colisiones, las cuales se deben minimizar. Se exponen una plétora de opiniones de expertos en la materia que justifican la necesidad de practicar pruebas para el avance del conocimiento en la materia. Se seleccionan tres algoritmos de criptografía liviana basados en la construcción esponja denominados Photon, Quark y Spongent, presentando sus principales características y desarrollando una implementación en C# de cada uno. A priori, se considera al primero como el más eficiente de los algoritmos seleccionados. Se evalúan realizando sucesivas pruebas de ejecución en un procesador de alto rendimiento, así como también en un dispositivo IoT, para registrar los tiempos empleados en obtener el digesto. Se miden la cantidad de instrucciones utilizadas y la memoria consumida.

Palabras clave: *Criptografía liviana; dispositivos conectados; Internet de las Cosas; Sistemas ciberfísicos; funciones hash livianas; Photon; Quark; Spongent; Construcción esponja; C#.*

Abstract

Internet of Things systems are scarce-resource devices that need to be protected with cryptography designed with their resource constraints in mind. Such cryptography is called "lightweight cryptography" and this project addresses the branch dedicated to hash functions. It has many characteristics such as: processing a message of arbitrary length and obtaining a pseudo-random output of fixed length; such a function should be easy to compute, but difficult to reverse. Since the length of the output is finite, the process is prone to collisions, which must be avoided. A plethora of opinions of experts in the field are exposed that justify the need to practice tests for the advancement of knowledge in the field. Three lightweight cryptography algorithms based on the sponge construction named Photon, Quark and Spongent are selected, presenting their main characteristics and a custom development in C# programming for each one. A priori, the former is considered the most efficient one. Algorithms will be evaluated by carrying out successive execution tests in a high end processor as well as in an IoT device. Processing times, the amount of instructions used and memory consumption are measured.

Keywords: *Lightweight Cryptography; IoT; Cyberphysical systems; lightweight hash functions; Photon; Quark; Spongent; Sponge construction; C#.*

CONTENIDO

1. Introducción

Con el advenimiento de internet, los bajos costos en la producción de procesadores y la posibilidad de tener poder de cómputo en objetos que superan las definiciones de "computadora", se acuñó el término de "Internet de las Cosas" o simplemente IoT por sus siglas en inglés "Internet of Things". Antiguamente se denominaba un "Sistema Embebido" a aquel sistema incorporado a otro de mayor jerarquía, en especial, a los dispositivos como microcontroladores, pequeñas computadoras comprendidas por un procesador y memoria ambos integrados en una sola pieza. Hoy en día, existe una amplia gama de estos sistemas. Entre ellos, podemos destacar las microcomputadoras de bajo costo como Arduino y Raspberry Pi, también denominadas "System on a Chip" por estar todos los componentes ensamblados en un único circuito integrado.

En la década de los 90, surgió el concepto de "computación omnipresente", sugiriendo la inminente masividad con la que los procesadores estaban alcanzando un público inimaginado en aquel entonces. La idea de que podrían diseñarse aparatos que procesaran información y estuvieran interconectados, más allá de los "Mainframes", y computadoras de escritorio ya merodeaba en la sociedad. Las redes inalámbricas ya eran un hecho. Faltaba poder explotar el poder del procesamiento interconectado.

El término Internet de las Cosas fue introducido en 1999 (Ashton 2009) en referencia a la posibilidad de implementar identificadores de radiofrecuencia RFID en la cadena de suministro de una empresa, y mediante sensores, las computadoras podrían recolectar una variedad de datos que luego podrían procesar en información. Lo importante aquí es destacar que esto lo realizarían sin la intervención del humano. A partir de ahora, el sistema de cómputo deja su rol pasivo, para meterse entre las tareas cotidianas para observar y comprender el mundo que lo rodea.

De esta manera, se crea el concepto de comunicación "donde sea, cuando sea". (Ibarra-Esquer et al. 2017) No muy lejos está la visión de Nikola Tesla, al afirmar que la misma tierra se convertiría en un "cerebro" enorme, al aplicar las, por entonces, teorizadas "redes inalámbricas" (Kennedy 1926). Las sociedades ponen como objetivo el bienestar colectivo. Y un camino en esa dirección se lograría al contar con sistemas que continuamente monitoreen nuestra salud, el medio ambiente, y los artefactos que utilizamos a diario, con el fin de recabar

información que esté inmediatamente disponible para poder tomar decisiones. De esta manera, estos aparatos deben mantenerse presentes, pero al margen. Activos, pero fuera de vista.

Existen varios términos que sugieren definir los mismos conceptos, tales como "Sistemas Embebidos", o "Sistemas Ciberfisicos". Éstos traen aparejados conceptos inherentes a su desarrollo, como el tiempo de repuesta, energía consumida y espacio que ocupan. De acuerdo con el National Research Council de los Estados Unidos, *"estamos por presenciar una revolución en los sistemas. Los objetos conectados tienen el poder de cambiar radicalmente la forma en que la gente interactúa con el medio ambiente, interconectando sensores y dispositivos que permiten recolectar, compartir y procesar información de manera nunca antes vista"* (Marwedel 2010).

Un ejemplo de estas tecnologías es el de una red energética inteligente, capaz de autoadministrar la producción y distribución de energía por medio de sensores y procesadores distribuidos por toda la cadena productiva, realizando cálculos para adaptarse a las demandas que sean requeridas, evitando el desperdicio de recursos. Y tantos otros más, capaces de interactuar con el humano a diario, ya sean en prótesis capaces de interpretar señales cerebrales para traducirlas en manipulación de objetos, hasta automóviles que conducen sin la intervención humana.

Algunas características ineludibles de estos objetos conectados los definen precisamente en su rol de estar permanentemente conectados a una red global, cualquiera sea su tipo. No es posible pensar un objeto conectado sin una red que lo abastezca. También se reconoce que deben interactuar con el mundo real. No solo deben procesar datos, sino también poder tomar decisiones y actuar en consecuencia. Esto requiere de cierta autonomía para poder ejecutar una acción, y desde luego, autonomía para poder mantenerse funcionando aun sin suministro de energía. Se espera que puedan relacionarse con otros sistemas autónomos, por lo que deberán presentar interfaces de control que permitan la interconexión entre ellos. Y, de esta manera, la información que puedan recolectar debe estar libremente disponible para ser compartida por varios sistemas a la vez, en una especie de repositorio común.

Debido a la naturaleza omnipresente de estos objetos conectados, la seguridad es una prioridad. Se espera que estos sistemas recolecten una cantidad de datos sensibles, los cuales deben ser administrados y transmitidos de forma segura. Debemos poder confiar en estos dispositivos, aun cuando no sepamos exactamente qué es lo que hacen. Según (Atzori, Iera y Morabito 2016), solo cuando estos sistemas sean lo suficientemente robustos, resilientes y

seguros, estarán tan insertos en la sociedad a tal punto que la gente podrá desestimarlos, en el sentido de otorgarle su total confianza, por el rol irremplazable que le será conferido en las distintas áreas de aplicación.

En la Ciudad Autónoma de Buenos Aires, en el año 2021, el presente Proyecto Final de Ingeniería tiene como objetivo analizar la criptografía liviana, la rama de la criptografía dedicada a brindar seguridad a los dispositivos conectados, y explicar la necesidad de contar con esta disciplina. En particular, se analizarán las funciones hash livianas, eligiendo los mejores criterios de evaluación aplicados al desarrollo de un "Framework" o ambiente de pruebas que permitan analizar su performance.

2. Antecedentes

2.1 Marco teórico

2.1.1 Generalidades de la criptografía

La criptografía es el estudio que busca dar seguridad a las comunicaciones, datos almacenados y sistemas en línea. Su objetivo es mantener en resguardo información confidencial, brindando acceso solamente al usuario autorizado (Paar y Pelzl 2009).

Los objetivos de la criptografía son los siguientes:

- *Confidencialidad*: Consiste en asegurar que solo el remitente indicado pueda descifrar un mensaje y leer su contenido.
- *No repudio*: El emisor del mensaje no puede negar haber enviado el mensaje en un futuro.
- *Integridad*: Garantizar que el receptor pueda discernir si el mensaje ha llegado tal cual ha sido emitido, o si ha sido modificado en su camino.
- *Autenticidad*: Asegurar que, tanto el emisor como el receptor del mensaje, son quienes dicen ser.

La criptografía puede dividirse en tres grandes grupos:

Criptografía Simétrica: Utiliza una única clave para cifrar un mensaje. Tanto el emisor como el receptor deberán contar con la misma clave para poder codificar y descifrar los datos. De igual manera, si los datos son almacenados localmente, se deberá contar con la clave utilizada para codificar y resguardar el acceso a los mismos. Ejemplos: DES (Davis 1978), AES (Daemen y Rijmen 1999).

Criptografía asimétrica: Utiliza dos claves, una para cifrar los datos y otra para descifrarlos. Una clave se mantiene en secreto y se denomina la "clave privada". La otra es de público conocimiento y puede ser utilizada por cualquier usuario. Es por esto que se la denomina la "clave pública". Ejemplos: Diffie-Hellman, Curvas Elípticas, Estándar de Firma Digital.

Este Proyecto Final de Ingeniería se centrará en el estudio de las Funciones Hash, la tercera rama de la criptografía.

2.1.2 Propiedades de una función hash

La *función hash* es una función que toma como entrada una cadena de caracteres de longitud arbitraria, la cual se denomina "el mensaje") y arroja como salida una cadena de bits de longitud fija. Un requerimiento básico es que los valores de *h(m)* sean fáciles de calcular, ya sea en implementaciones de software o hardware (Delfs y Knebl 2015).

$$h: \{0,1\}^* \rightarrow \{0,1\}^n , m \rightarrow h(m) \quad (1)$$

Siendo la longitud de *n* generalmente entre 128 y 512 bits.

Es un aspecto clave y de discusión la elección de la longitud de *n*, es decir la longitud del digesto obtenido. En teoría de la probabilidad existe un problema que es de utilidad para representar este dilema. La llamada "paradoja del cumpleaños" consiste en que en un conjunto de personas de tamaño *n* elegidas al azar, es probable encontrar una pareja que comparta el mismo día de cumpleaños. Lo llamativo es que la probabilidad supera el 50% para un grupo de tan solo 23 personas, y alcanza valores cercanos al 100% cuando el grupo asciende a tan solo 70 personas. En criptografía se denomina "ataque del cumpleaños" al ataque de fuerza bruta que intenta someter la resistencia a colisiones de una función hash. Consiste en generar tantos mensajes *m* como se pueda. Para cada mensaje, se computa su correspondiente *h(m)* y lo compara con los anteriores digestos obtenidos. Se espera poder encontrar una colisión con $2^{n/2}$ mensajes. De esta manera, es necesario proponer un *n* lo suficientemente grande para que sea computacionalmente imposible calcular y almacenar esa cantidad de digestos, es decir que se pueda calcular pero que requeriría demasiados recursos para llevarlo a cabo (Henderson 2013).

En un esquema de firma digital donde un mensaje es firmado junto con su digesto, el "ataque del cumpleaños" puede llevarse a cabo de la siguiente manera: Para dos mensajes distintos m_1 y m_2 se requiere obtener un mismo digesto para ambos. El atacante pretende crear

un m_2 tal que parezca haber sido firmado por el emisor, pero cuyo contenido no haya sido emitido por este último. En caso de obtener dicho digesto, el emisor no podría negar haber firmado el mensaje m_2, puesto que ambos derivan en el mismo digesto. Ahora, sabemos que el más mínimo cambio a un mensaje produce digestos enteramente diferentes. Pero m_1 y m_2 no necesariamente deben ser del todo distintos. Por ejemplo, se pueden crear dos archivos PDF con el mismo texto e imágenes. Con tan solo variar un pixel de una imagen del archivo PDF, estaríamos obteniendo distintos digestos para cada variación. Ese cambio puede ser imperceptible al ojo humano, como puede ser intercambiar el pixel por otro de un color muy similar, y permitiría generar un digesto idéntico al obtenido a partir de un documento original pero para un documento apócrifo. De similar forma, es lo que un equipo de seguridad de Google logró realizar al quebrantar el algoritmo SHA-1 (Marc Stevens et al. 2017).

Funciones de "un solo sentido": Las funciones hash tienen un uso típico que es el cifrado de contraseñas. Esto puede servir para almacenar el valor $h(clave)$ en una base de datos, en vez guardar directamente la *clave*. Al ingresar el usuario su contraseña, se calcula la función hash de la misma y se compara con el valor almacenado. Esta técnica se denomina "cifrado irreversible" y se utiliza en los sistemas operativos para impedir que los administradores de las bases de datos tengan acceso a las contraseñas de los usuarios si las mismas se almacenan en texto plano, ya que no es posible calcular la clave a partir de su valor obtenido con $h(clave)$. Por eso se denominan que son de "un solo sentido", ya que dado un valor $y \in \{0,1\}^n$ es computacionalmente improbable encontrar un m con un $h(m) = y$.

Resistencia a la segunda preimagen: Las funciones hash se utilizan en los modelos de firma digital. Un mensaje a ser transmitido se le aplica una función hash y luego ese digesto es firmado en lugar del mensaje original. De esta manera, un adversario que quiera hacerse pasar por el emisor original del mensaje no debería tener posibilidad de encontrar un mensaje tal que $m' \neq m$ con $h(m') = h(m)$ siendo m el mensaje original y $h(m)$ ya conocido de antemano.

Resistencia a las colisiones: Una colisión ocurre cuando, partiendo de dos mensajes distintos, se obtiene un mismo digesto. Dado que la longitud del digesto es finita, y los valores de entrada se consideran infinitos, es correcto deducir que la cantidad de colisiones es infinita. En todo caso, se considera que una función hash cumple con esta propiedad cuando la probabilidad de encontrar una colisión es muy baja y requiera de un amplio poder de cómputo.

Es decir, una colisión ocurre cuando se encuentra el par (m, m') con $m \neq m'$ y $h(m) = h(m')$. Si es computacionalmente imposible encontrar una colisión para una determinada función hash, entonces dicha función es resistente a las colisiones.

Para realizar un ataque de fuerza bruta a la resistencia de la segunda preimagen, teniendo un mensaje *m* y su correspondiente digesto se necesitaría generar distintos mensajes $m' \neq m$ y calcular $h(m')$ tal que ese digesto obtenido coincida con el primero $h(m)$. En el caso de la resistencia a la preimagen, un atacante que posee un determinado digesto *x* no debería poder encontrar un mensaje de entrada *m* tal que $x = h(m)$ antes de haber procesado 2^n mensajes. En este caso, no es necesario contar con un valor de *n* muy grande, como sí lo es para proteger la función hash del "ataque del cumpleaños" o la resistencia a colisiones.

Otros dos conceptos empleados en la construcción de una función hash son los de confusión y difusión, definidos por (Shannon 1949).

La propiedad de confusión radica en la complejidad subyacente al calcular el digesto con una clave, a partir de un mensaje dado (Coskun y Memon 2006). Debe volverse impracticable deducir una relación entre dos digestos *H* y *H'* obtenidos a partir de un mensaje *m,* una clave *k,* y una función hash *H,* tal que:

$$H = h(m, k) \quad (2)$$

$$H' = h(m, k') \quad (3)$$

Donde *k* y *k'* pueden diferir en solamente un bit. En una función con propiedades de confusión fuertes debe cumplirse que, de cambiar *k,* la probabilidad de que cada bit del digesto cambie de 0 a 1 o viceversa debe ser de 0,5 y observarse una distribución azarosa para aquellos bits cambiados. Para funciones hash débiles, se espera que la distancia de Hamming, es decir la cantidad de bits que fueron cambiados entre ambos digestos sea mínima, así como también es mínima dicha distancia entre las claves empleadas. De esta manera se obtienen digestos muy similares entre mensajes idénticos y claves levemente alteradas. Este tipo de función hash es susceptible al ataque de fuerza bruta.

La propiedad de difusión define la complejidad de relacionar el mensaje original con el digesto obtenido (Coskun y Memon 2006). Se espera que cada bit de entrada afecte a cada bit de salida, provocando un "efecto mariposa", donde pequeños cambios en el mensaje se propaguen de manera amplificada al digesto.

$$H = h(m, k) \quad (4)$$

$$H' = h(m', k) \quad (5)$$

Debe dificultarse obtener la relación entre *H* y *H'* donde *m* y *m'* difieren en solo un bit, ya que se espera que cada bit que cambie de 0 a 1 o viceversa en la entrada, afectará a cada bit de salida con una probabilidad de 0,5. De lo contrario, puede esperarse obtener colisiones fácilmente ya que se podrían predecir los cambios esperados en la salida a determinados cambios aplicados en la entrada.

Además, es importante destacar dos formulaciones de eminencias de la criptografía. La primera, emitida en el siglo XIX por el francés Auguste Kerckhoffs (Kerckhoffs 1883) que dicta "Un sistema criptográfico debería ser seguro aun cuando todo acerca del sistema es conocido, excepto la clave". La segunda, por el norteamericano Claude Shannon, cuya máxima dice "El enemigo conoce el sistema". Ambas premisas establecen que un sólido algoritmo criptográfico debe ser lo suficientemente bueno como para que, a pesar de conocerse sus operaciones internas, sea computacionalmente imposible quebrantarlo. Es importante destacar que los algoritmos más seguros en la actualidad cumplen con estas premisas, ya que son de código abierto. Por el contrario, todo sistema que utilice algoritmos propietarios y cerrados a revisión, se consideran por defecto inseguros.

2.1.3 Construcción de una función hash

Existen varias formas de abordar la construcción de una función hash:

Construcción Merkle-Damgård: Consiste en reducir el dominio de la función hash aplicando otra función denominada "función de compresión", la cual vincula un mensaje *m* de longitud $n + r$ con mensajes $f(m)$ de longitud n. Se denomina r a la proporción de compresión. Se asume que la función de compresión es resistente a colisiones, y por consecuencia la función hash también lo será.

Construcción Sponge o "esponja": Es una generalización de la anterior. Utiliza una función "esponja" en lugar de una función de compresión. Procesa cadena de caracteres binarios de cualquier longitud. Y la longitud de la salida obtenida puede modificarse para adaptarse a las necesidades de la aplicación. Procesa el mensaje en trozos de cierta longitud *r* y la salida es generada en trozos de la misma longitud. Finalmente, el digesto es obtenido iterativamente aplicando una permutación. Se lo denomina "esponja" porque en el proceso interno iterativo, existe una fase de "absorción" donde se procesa el mensaje de entrada en una cierta cantidad de rondas, para luego pasar a la fase de "exprimido", donde se genera la salida.

2.1.4 Esquema general de AES

Dado que algunas funciones hash a analizar comparten metodologías de procesamiento de bytes para obtener el digesto partiendo de técnicas empleadas por el algoritmo AES, es necesario mencionar algunos conceptos relevantes.

El algoritmo AES (Estándar de Encriptación Avanzada) o también conocido por su nombre original Rijndael, es un estándar de cifrado reconocido por el Instituto de Estándares y Tecnología de Estados Unidos (NIST) en 2001 (Wehbe 2021b). El algoritmo es el sucesor de DES (Data Encryption Standard) publicado en los ´70 (Davis 1978). Es un algoritmo simétrico, ya que la misma clave es usada para cifrar y descifrar los datos. Una serie de operaciones es aplicada a un mensaje o cadena de entrada para obtener una cadena cifrada. Para volver a obtener el mensaje original, basta con repetir las mismas operaciones en el camino inverso.

AES (Daemen y Rijmen 1999) funciona aplicando distintas etapas en varias rondas n_r, cuya cantidad depende del tamaño de las claves. Para el caso de una longitud de 128 bits se emplearán 10 rondas; si la longitud de las claves es de 192 bits, la cantidad de rondas asciende a 12. Y para clave de 256 bits, el tamaño de $n_r = 14$ rondas.

Los datos binarios empleados en algoritmos criptográficos permite expresar una sucesión de bits como un campo finito o "Campo de Galois", donde existen una cantidad finita de elementos en este caso representados por secuencias de 0 y 1, los componentes del campo. Esta representación permite mezclar los datos de manera sencilla y efectiva (Benvenuto 2012).

Para obtener la representación binaria de un número decimal, se realiza sumando términos de la forma:

$$x = \sum_{n \ni \mathrm{N}} a_n 2^n \tag{6}$$

Por ejemplo, la representación del número 19 está dada por la sucesión de sumas:

$$19 = \ldots + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \tag{7}$$

Siendo la representación en binario: 10011, ó 00010011 como en realidad se almacena un byte, el cual es un campo de Galois de orden 2^8, o $gf(2^8)$.

La operación de adición y substracción devuelven el mismo resultado cuando el campo característico es 2, y su valor máximo no supera el valor 11111111 o 255 en decimal, el valor más grande que un byte puede almacenar. Dichas operaciones pueden reemplazarse por la de operación lógica de Ó exclusivo o XOR, la cual devuelve 0 si ambas entradas son iguales, o 1

si ambas son diferentes. También se utiliza la operación de multiplicación, que involucra una operación de "resto" o *modulo* con un polinomio irreducible.

En AES, existen tres tipos de capas que conforman el proceso, junto con una matriz de estado, la cual es utilizada para cifrar los datos en diferentes etapas. Las capas se describen a continuación:

Capa de Adición de Clave: una clave de ronda, o subclave la cual ha sido derivada de una tabla de claves, es aplicada la operación XOR al estado actual.

Existen dos entradas para esta capa. Una de ellas consiste en el estado actual del vector de 16 bytes, y la otra es una clave derivada la cual también consiste en 16 bytes (o 128 bits). A ambas se le aplica la función XOR (lo que equivale a la adición en un Campo de Galois, antes mencionada).

Para una clave de longitud de 128 bits, existe 11 claves las cuales están almacenadas en un arreglo denominado "de expansión de clave". Cada celda posee cuatro bytes. La sub clave para la ronda *i* se crea a partir de los elementos:

$$W[4i], W[4i+1], W[4i+2], W[4i+3] \tag{8}$$

La ronda inicial toma como entrada la clave inicial, y cada ronda hasta la décima ronda subsecuente toma como entrada el resultado de la ronda anterior. Todos los elementos, a excepción del primero se obtienen recursivamente como sigue, donde $\oplus$ es el operador XOR:

$$W[4i + j] = W[4i + j - 1] \oplus W[4(i - 1) + j] \tag{9}$$

$$\text{Con } (i = 1, \dots, 10) \text{ y } (j = 1,2,3)$$

Y para el primer elemento de la izquierda, se obtiene:

$$W[4i] = W[4(i - 1) + g(W[4i - 1]) \tag{10}$$

$$\text{Con } i = 1, \dots, 10$$

Donde la función g es una función no lineal cuya entrada y salida es una secuencia de cuatro bytes. Dicha función rota la entrada de bytes, luego aplica una función S-Box a cada uno y agrega al byte izquierdo más significativo un "coeficiente de ronda".

S-BOX, o capa de sustitución de bytes: cada elemento del vector estado es transformado utilizando tablas precalculadas con propiedades particulares. Este es el único paso no lineal del algoritmo. Esta operación se aplica en todas las rondas, exceptuando la inicial. En esta capa, existen 16 cajas de sustitución paralelas, cada una procesando un byte de entrada. Debe cumplirse que cada byte es reemplazado por otro byte:

$$B_i = S(A_i) \tag{11}$$

Debe ser no lineal, de la forma:

$$S(A) + S(B) \neq \mathrm{S(A + B)} \tag{12}$$

Debe ser biyectiva para poder ser reversible, y cada byte estar mapeado con otro byte. Además, no existen puntos fijos tales que no existan valores de A de la manera:

$$S(A) = A \tag{13}$$

En la primera etapa, se computan los inversos multiplicativos utilizando una matriz prediseñada con propiedades matemáticas de los campos extendidos. La segunda etapa consiste en multiplicar cada byte obtenido por una matriz de bits constantes, seguida por la adicción de un vector de 8 bits contante.

Las cajas denominadas S-boxes casi nunca son explícitamente computadas. Usualmente se implementan en software como una tabla de consulta directa que entrega el resultado en cada operación sin realizar cálculo alguno.

Capa de Difusión: Esta capa contiene dos subcapas. El objetivo de esta capa es esparcir la influencia de bytes individuales a través de todo el vector estado. A diferencia de las S-boxes, esta es una transformación lineal en el sentido de que la transformación que surge de la adición de dos bytes es la adición de la transformación individual de cada byte. A continuación, las dos subcapas:

- **Corrimiento de Filas:** Aquí se realiza una permutación del vector estado a nivel de byte. El corrimiento fortalece el proceso de propagar la influencia de cada byte individual a todo el vector estado.
- **Mezclar Columnas:** Es una operación de matrices que combina las columnas de cuatro bytes en una. Es la operación más relevante, ya que permite aplicar difusión a través de todo el vector estado.

La primera ronda consiste únicamente en la capa de Adición de Clave. Luego, las rondas 1 a $n_r - 1$ consisten todas de tres capas. En la última ronda, se ejecutan las tres capas con la diferencia de que solo se aplica la función Mezclar Filas. Además, existe una operación XOR con una clave derivada al comienzo y al final de todo el proceso, referida usualmente como el "blanqueo de clave".

2.1.5 El algoritmo Sponge

Obtiene su nombre de la palabra "esponja" en inglés, haciendo referencia a la capacidad de absorción y devolución de bytes por parte de esta función (Bertoni et al. 2011). El algoritmo provee una manera de generalizar las funciones hash, permitiendo digestos cuyo tamaño no es fijo. Consiste en la construcción de una función con entrada y salida variables basada en una permutación de longitud fija. De esta manera, el algoritmo también puede ser empleado como cifrador de flujo. Dada la robustez con la que fue concebido, permite emplearlo como un "oráculo aleatorio", concepto que idealiza un algoritmo perfecto el cual mapea cada entrada con una salida totalmente azarosa, pero con la condición de devolver siempre la misma salida a igual entrada. La limitación del algoritmo Sponge radica en la finitud de la memoria empleada.

El algoritmo está definido por un estado, que consiste de dos secuencias de bits (Wehbe 2021a):

- Capacidad (*m* bits)
- Tasa de bits (*r* bits)

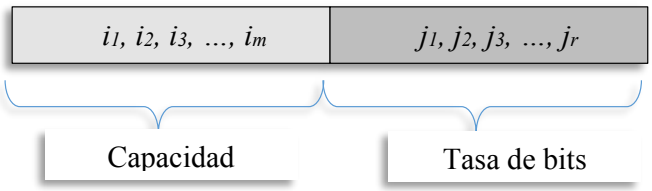


Figura 1: La concatenación de capacidad y tasa de bits da lugar al estado. (Wehbe, 2021)

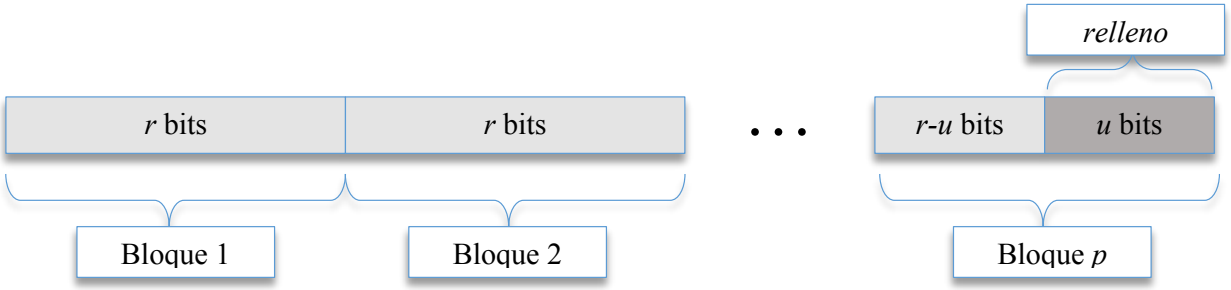


Figura 2: El número r es el tamaño de los bloques en que se divide la cadena de caracteres al que se le va a aplicar la función de hashing. (Wehbe, 2021)

En Photon (Guo, Peyrin y Poschmann 2011), el relleno o *padding* se hace agregando un 1 seguido del número de ceros necesario. Como siempre en el caso de *padding*, si el mensaje tuviera un múltiplo de *r*, debe agregarse un bloque de *padding* para evitar ambigüedad.

Esto se complementa con una función que es una permutación pseudo-aleatoria de los $m+r$ bits del estado:

$$f: \{0,1\}^{m+r} \rightarrow \{0,1\}^{m+r} \quad (14)$$

El esquema general es el siguiente:

1. El estado es inicializado en 0.
2. Para cada bloque i:
    a. Se calcula $\mathbf{bitrate_{i+1} \leftarrow bloque_i \oplus bitrate_i}$
    b. Se efectúa $\mathbf{estado_{i+1} \leftarrow f(estado_i)}$

Esta fase se denomina *absorción*, pues los p bloques son absorbidos por la esponja, produciendo un resultado final de $m+r$ bits (el estado p). La salida se produce "estrujando" la esponja. El proceso para producir una salida de q bits (donde q es un múltiplo de un número r', es decir $q = t \cdot r'$) se describe a continuación. La salida consiste en t bloques de r' bits. Llamamos s a la función que nos devuelve los últimos r' bits de una cadena de $m+r$ bits. Es decir, ingresamos el estado y nos devuelve los r' bits de la derecha:

$$s: \{0,1\}^{m+r} \rightarrow \{0,1\}^{r'} \quad (15)$$

Entonces:

- Repetir para $j = 0$ a $t-1$
    - **Salida = $s(estado_{p+j})$ || Salida**
    - **$Estado_{p+j+1} \leftarrow f(estado_{p+j})$**

Aquí se "estruja" de una esponja de $m+r$ bits una salida de q bits.

2.2 Estado del Arte

La investigación y desarrollo de algoritmos de criptografía liviana se encuentra en pleno progreso. Ya se ha identificado la necesidad de contar con algoritmos diseñados específicamente para dispositivos de bajas prestaciones. En la década pasada se han propuesto diversos algoritmos que aún no se han convertido en estándares.

Una investigación publicada por (Hammad et al. 2017) donde hace un relevamiento de las funciones hash desarrolladas con el propósito de ser utilizadas en dispositivos de escasos recursos y define criterios de evaluación para poder compararlas. Las funciones seleccionadas por el autor son: Spongent, Photon, Gluon, Armadillo, DM-Present y H-Present, entre otras.

Un artículo publicado por (Buchanan, Li y Asif 2017) hace un amplio relevamiento del estado de situación. La criptografía liviana intenta reducir el alto poder de procesamiento y

espacio en memoria requerido para entornos estándares. Consiste en reducir el tamaño de las claves, la tasa de rendimiento, energía consumida y el área medida en GE por *Gate Equivalence* o Equivalencia de Compuertas.

El NIST y la Organización Internacional ISO/IEC definen un cierto número de métodos a emplear en aparatos IoT y RFID, y distingue dos tipos de criptografía: la convencional, utilizada en servidores, estaciones de trabajo, tabletas y celulares; y la criptografía liviana, empleada en sistemas embebidos, RFID y redes de sensores.

Dado que los sistemas embebidos poseen generalmente microcontroladores de 32-bits, 16-bits, 8-bits y hasta de 4-bits, estos no pueden llevar a cabo la criptografía convencional.

Es una dificultad utilizar el estándar AES, ya que en dispositivos embebidos es común observar tamaño de bloques más pequeños, de entre 64 y 80 bits, claves más cortas de menos de 90 bits, y ciclos menos complejos. A esto se suma que presentan vulnerabilidades en ataques de canal lateral, que consiste en correlacionar el consumo de energía del dispositivo para inferir las operaciones realizadas, o que permiten reconstruir el estado secreto inicial a partir de una captura del estado interno intermedio alojado en la memoria volátil del ordenador.

Las principales restricciones en el diseño de algoritmos criptográficos livianos están relacionados al consumo de energía, cantidad de compuertas y tiempo de procesamiento. En los dispositivos pasivos, como los RFID, no existe una batería que los alimente. El propio chip debe alimentarse de la energía asociada a la onda de radio recibida. Y para aquellos dispositivos activos, recargar la batería puede ser una tarea no habitual. Por lo que las funciones empleadas deben estar optimizadas para drenar la batería lo menos posible.

Para las funciones hash en general, hay que considerar que los procesadores empleados en dispositivos IoT están valuados en centavos de dólar estadounidense, mientras que un procesador de computadora de escritorio lo hace en cientos de dólares. Es por esto que se descartan las funciones más utilizadas en la criptografía tradicional, como ser MD5 y SHA-1 por no ser lo suficientemente eficientes para dispositivos IoT. El NIST ha recomendado nuevos métodos denominados SPONGENT, PHOTON, Quark y Lesamnta-LW, los cuales tienen un impacto en la memoria mucho menor, de solo 256 caracteres de entrada, cuando una función tradicional lo hace en el orden de los 2^{64} bits.

El NIST (Computer Security Division y NIST 2017) está realizando desde el año 2015 y actualmente en desarrollo una convocatoria para solicitar, evaluar y estandarizar algoritmos de

criptografía liviana que sean apropiados para ser empleados en entornos de pocos recursos, donde los estándares de criptografía actuales definidos por este instituto no son aceptables. Actualmente los algoritmos candidatos ya han sido presentados, encontrándose la convocatoria en este mismo momento en la etapa final. Los algoritmos que llegaron a esta instancia en marzo de 2021 son: SCON, Elephant, GIFT-COFB, Grain128-AEAD, ISAP, Photon-Beetle, Romulus, Sparkle, TinyJambu, y Xoodyak. La ronda final espera ser evaluada en un período aproximado de 12 meses. Existen cinco equipos encargados de realizar las pruebas de rendimiento de estos algoritmos, con distintos entornos de desarrollo según la arquitectura del micro controlador.

Todavía no existe un estándar definido por el NIST para suites de algoritmos de criptografía liviana. Probablemente el año entrante habrá novedades respecto a la selección que proponga dicho instituto, lo cual será un hito alcanzado en el ámbito de la investigación de algoritmos que satisfagan las necesidades de estos dispositivos conectados.

2.3 User Research

Aquí se recopilan diversas declaraciones de expertos en criptografía que resaltan la importancia de avanzar en la búsqueda de algoritmos que puedan utilizarse en objetos conectados y de pocos recursos, y la relevancia de realizar una investigación, tal como lo es este presente proyecto.

Según (Buchanan, Li y Asif 2017), existe una compensación de costo-beneficio al diseñar un algoritmo o función para sistemas de pocos recursos. Teniendo en cuenta el costo monetario, la velocidad, seguridad y performance, existe una motivación para diseñar estas herramientas que cumplan con la premisa de usar menos memoria, menos recursos del procesador y menos energía sin comprometer la seguridad brindada, llevándola a un nivel aceptable que, de otro modo, no sería alcanzable.

Para (Mouha 2015), lo más importante para la criptografía liviana está pensada para aplicaciones con estrictos requisitos. Se debe enfocar precisamente en aplicaciones donde la criptografía sea el cuello de botella. La criptografía convencional no fue pensada con una optimización en mente. Esto no quiere decir que sea ineficiente, pero su diseño no contempló recursos limitados, ya que apuntaba a una amplia gama de dispositivos sin notables restricciones. Para la criptografía liviana, su fin debe ser realizar algoritmos hechos “a medida” para cada caso de uso, con el costo de ser aplicados en una menor gama de dispositivos. La

clave debe ser enfocarse en el diseño iterativo. Comenzar el diseño con pocas restricciones, y gradualmente ir ajustando distintos parámetros hasta alcanzar las restricciones impuestas por el determinado caso de uso. Finalmente, el autor se plantea preguntas que este Proyecto Final de Ingeniería intenta analizar. Tales como: ¿Cuáles son las restricciones en el tamaño del código? ¿Cuánta memoria RAM hay disponible? ¿Cuáles son las restricciones de latencia, rendimiento, y energía consumida?

La convocatoria del NIST para desarrollar algoritmos de criptografía liviana deja en claro manifiesto la necesidad mundial de contar con algoritmos capaces de operar en sistemas de pocos recursos. El objetivo final es desarrollar estándares de los cuales se pueda beneficiar el mercado entero. Según la Dra. Kerry McKay: "*Los estándares traerán una solución bien definida que aplique a una amplia gama de situaciones. Los aparatos IoT están en clara expansión, pero la mayoría de ellos presenta nula seguridad. Son tan diversos los dispositivos y casos de uso que es difícil clasificarlos a todos. Hay muchísimos ataques posibles, de distintas variantes. Debemos ser amplios para abarcarlos a todos.*" (Swenson 2018).

En especial referencia a las funciones hash, McKay afirma que las funciones hash deberá compartir recursos con el proceso de encriptación simétrica AEAD, cifrado autenticado con datos asociados, que permite revisar la integridad de la parte cifrada y no cifrada del mensaje.

Por su parte, el Dr. Bruce Schneier, autor de un popular Blog de ciberseguridad, comenta incidencias en la convocatoria del NIST para la selección del algoritmo conocido como SHA-3. La función hash de dicho conjunto de algoritmos, denominada Keccak, sufrió alteraciones que redujeron su nivel de seguridad, en favor de garantizar una mejor performance. Coincidentemente con las opiniones de Edward Snowden (Verble 2014), quien acusara al organismo de seguridad nacional NSA de intentar manipular los estándares criptográficos con los fines de empobrecerlos para poder explotarlos a su favor, cualquier modificación que sufriera el estándar seleccionado como ganador levantaría sospechas. "*Hay demasiada desconfianza en el aire. NIST se arriesga a publicar un algoritmo en el que nadie confiará, y que nadie (excepto aquellos obligados) usará"* (Schneier 2013). El riesgo que existe en no tener un adecuado escrutinio de los algoritmos criptográficos socava la confianza en ellos.

De esta manera, los autores convergen en la necesidad de continuar con la investigación y desarrollo de la criptografía liviana, área que todavía está en evolución. Estos son fundamentos muy importantes para realizar el presente Proyecto Final de Ingeniería.

3. Hipótesis

En el presente proyecto de investigación se busca comparar diferentes algoritmos de funciones hash de criptografía liviana. Luego de una preselección de posibles algoritmos a evaluar, se puede destacar el diseño empleado en la función denominada PHOTON (Guo, Peyrin y Poschmann 2011). De esta manera, se selecciona este algoritmo como base de pruebas y se plantea la hipótesis siguiente:

"El algoritmo de función hash PHOTON es el más eficiente de los algoritmos de funciones hash para criptografía liviana seleccionados".

4. Metodología

El presente proyecto tiene como finalidad realizar una implementación en software de los distintos algoritmos a evaluar, y desarrollar un *framework* teórico de pruebas, definiendo distintas métricas y características de relevancia para efectuar las comparaciones. El lenguaje de programación elegido es C#, mediante el uso de la plataforma Visual Studio 2019.

Se comienza creando una solución, donde se incorpora un proyecto programado en C#. El proyecto consta de tres archivos denominados Constants.cs, (nombre-función-hash).cs y Program.cs.

El archivo Program.cs contiene el programa conductor principal o *driver*. El archivo denominado con el nombre de la función hash contendrá los algoritmos empleados para obtener el digesto, mientras que el archivo Constants.cs contendrá todos los valores que permanecen inalterados en todo el transcurso de ejecución de la función hash.

Es importante destacar que existen diferentes configuraciones para un mismo algoritmo, donde cada configuración explicita sus propias constantes previo a la compilación del mismo.

Si bien el lenguaje C# permite el uso de clases en programación orientada a objetos, la implementación del algoritmo fue realizada tratando de mantener el esquema de programación procedural, contenido en una sola clase denominada *Program* y separado en distintos archivos mediante la posibilidad de fragmentar clases en diferentes archivos a través del modificador de clase *partial*. Se trató de minimizar el uso de librerías externas para disminuir el impacto en la memoria empleada. Tampoco se utilizó código denominado "inseguro", que permite administrar el uso de la memoria.

A continuación, se exponen los procedimientos internos de las funciones hash seleccionadas a partir del documento publicado por los respectivos autores y en base al código implementado.

4.1. Photon

La función hash denominada Photon (Guo, Peyrin y Poschmann 2011) o "fotón", como la partícula elemental sin masa que transporta la luz y viaja a la máxima velocidad alcanzable en el vacío, probablemente en alusión a lo compacto, velocidad de ejecución y eficiencia del algoritmo empleado. Fue pensado para implementarse en hardware, en especial en etiquetas RFID. Para lograr consumir un reducido espacio de memoria, se eligió emplear la estructura de un algoritmo de esponja. Las permutaciones empleadas están basadas en el algoritmo AES, mezclando columnas en serie.

A continuación se detallan las constantes y funciones empleadas en el algoritmo, en su configuración PHOTON-80/20/16, según (Guo, Peyrin y Poschmann 2011) y su implementación original (Guo 2020):

4.1.1. Constantes

ROUND: cantidad de rondas a realizar. Valor: 12.

S: Tamaño en bits a utilizar. Será de 4 para todas las configuraciones, excepto en las variantes Photon256 y PhotonAes, donde el valor es de 8.

D: Define la dimensión del vector estado. Valor: 5.

RATE: Cantidad de bits de entrada a procesar por iteración. Valor: 20

RATEP: Cantidad de bits de salida a procesar por iteración. Valor: 16.

DIGESTSIZE: Tamaño del digesto final. Es un valor expresado en bits que representa la longitud final del digesto. En el caso de la configuración con un valor de 80, representa 10 caracteres hexadecimales como resultado de digesto, donde cada dígito hexadecimal representa 4 bits (o medio byte, unidad también conocida como *nibble*).

RC: Matriz de doble entrada denominada *Round Coefficient*, con valores precalculados. La cantidad de filas debe ser igual a D, y la cantidad de columnas igual a la cantidad de rondas.

WORDFILTER: Corrimiento de bytes precalculado tomando como base la constante S.

SBOX: Caja de sustitución de bytes predefinida en un arreglo de 16 posiciones.

MixColMatrix: Matriz cuadrada de dos dimensiones definida ambas por el valor de D.

ReductionPoly: Valor hexadecimal 0x3 utilizado en una función XOR.

DEBUG: Bandera para permitir mostrar por consola los estados internos del proceso.

4.1.2. Funciones

El archivo Program.cs es el archivo conductor, conteniendo la función *Main* donde se hacen los llamados a las funciones *Hash* y *PrintDigest* para aplicar la función hash y mostrar el digesto obtenido. El archivo Photon.cs es el que contiene todas las funciones relevantes, las cuales son llamadas por la función *Hash* y sus subfunciones.

La función Hash recibe el arreglo del digesto por referencia, además de la cadena de caracteres o mensaje y la longitud de dicha cadena expresada en bits. Se tiene cuenta que en el lenguaje C# los vectores son siempre pasados por referencia, y esta, a su vez, es pasada por valor. Es decir que la función trabajará con el mismo vector, a menos que se realice una nueva asignación y se pierda esa referencia. En ese caso, los cambios no se verán reflejados por fuera de la función. En caso de pasar el vector con la palabra clave *ref*, se podrá reasignar el mismo objeto a una nueva instancia con los cambios reflejados por fuera del método llamado.

Se procede a definir dos arreglos. Uno de ellos denominado *state* de dos dimensiones, con la constante definida como D. El otro, denominado *padded* o relleno que consta de una dimensión y de 4 posiciones.

La función **Hash** está estructurada de la siguiente manera:

1. Inicialización del vector bidimensional de estado.
2. Bucle WHILE donde se procesa el mensaje con una determinada tasa de proceso definida por la constante RATE, aplicando la función de Compresión al mensaje.
3. Dos bucles FOR secuenciales para inicializar el vector de relleno.
4. Llamado a la función de Compresión.
5. Llamado a la función de estrujamiento o *Squeeze*.

Función **Init**: Se omite la inicialización del vector estado a cero, ya que la misma es realizada automáticamente al crear un nuevo vector en C#. Se define un vector de 3 posiciones y se lo inicializa con tres operaciones de AND a nivel bit, utilizando las constantes DIGESTSIZE, RATE y RATEP y el valor hexadecimal 0xFF. La inicialización finaliza llamando a la función WordXorByte.

Función **WordXorByte**: En un bucle WHILE se realiza la operación XOR.

Función **GetByte**: Función auxiliar de WordXorByte para obtener un byte a través de operaciones de corrimiento de bits y una operación de AND junto con la constante definida como WORDFILTER.

Función **CompressFunction**: Función de Compresión. Consiste en aplicar la operación XOR y permutación de bytes en dos llamados secuenciales a WordXorByte y Permutation.

Función **Permutation**: Mediante un bucle FOR realiza la cantidad de rondas definidas en la constante R, aplicando al vector estado las funciones secuenciales AddKey, SubCell, ShiftRow y MixColumn.

Función **Squeeze**: Mediante un bucle WHILE que interrumpirá su ejecución hasta alcanzar el tamaño del digesto definido en la constante DIGESTSIZE, llamará a la función WordToByte en todas las vueltas, y a la función Permutation en todas exceptuando la última vuelta.

Función **WordToByte**: Escribe los distintos bytes en el vector estado, haciendo uso de la función WriteByte.

Función **WriteByte**: Escribe en un vector el digesto final, aplicando operaciones de bits según una condición.

Existen diferentes configuraciones para la función hash Photon, cada una denotada con la expresión PHOTON-*n/r/r'* donde *n* es el tamaño final del digesto con valores entre 64 y 256 bits, *r* y *r'* son la tasa de bits de entrada y de salida respectivamente. El tamaño del estado interno $t = (c+r)$ depende del tamaño del digesto y puede tomar solamente cinco valores, 100, 144, 196, 246 y 288 bits. De esta manera, solo deben definirse cinco permutaciones internas P_t para cada estado interno. Para poder cubrir un amplio espectro de aplicaciones, los creadores de PHOTON proponen cinco variantes del algoritmo en base a dichos valores de permutaciones internas P_{100}, P_{144}, P_{196}, P_{256} y P_{288} para PHOTON-80/20/16, 128/16/16, 160/36/36, 224/32/32 y 256/32/32 respectivamente. La primera configuración se considera adecuada para casos donde una preimagen de 64 bits y código de autenticación de mensaje o MAC de igual tamaño son suficientes. En contraposición a la última configuración, que ofrece un alto nivel de seguridad al tener una resistencia a las colisiones de 128 bits.

El mensaje *M* a procesar es rellenado anteponiendo un 1 y tantos 0 hasta que la longitud total sea un múltiplo de la tasa de bit de entrada *r* y obtener *l* bloques de mensaje $m_{0},..,m_{l-1}$ de *r* bits cada uno. El estado interno *S* de *t* bits se inicializa estableciendo el valor S_0 = IV (vector

inicialización) = $\{0\}^{t-24}$ || *n/4* || *r* || *r'* donde || significa la concatenación de cada valor codificado en 8 bits, y cada byte es interpretado en la forma "big-endian", es decir el bit más significativo a la izquierda.

A continuación, se aplica el esquema de *esponja* clásico, como en la Figura 3, donde en cada iteración *i* se absorbe el bloque de mensaje m_i en la parte más a la izquierda del estado interno S_i y luego se aplica la permutación P_t. Una vez que todos los bloques hayan sido absorbidos, se construye el digesto concatenando sucesivos bloques de tamaño *r'* denominados z_i hasta alcanzar el tamaño de digesto *n* apropiado, de la forma $hash = z_0 \,||\ldots||\, z_{l'-1}$ donde *l'* denota el número de iteraciones de estrujes siendo *l'*=[*n/r'*]-1. Si el tamaño del digesto no es un múltiplo de *r'* simplemente se trunca $z_{l'-1}$ a *n mod r'* bits.

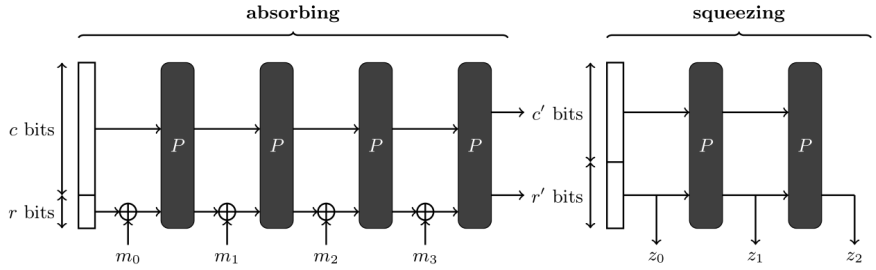


Figura 3: Esquema de un algoritmo esponja. (Guo et. al., 2011)

En la función dedicada a aplicar permutaciones se aplican operaciones de cuatro capas, como en la Figura 4. El estado interno de la permutación es una matriz cuadrada de *d x d* con celdas de tamaño *s,* donde siempre será de 4 bits a excepción de la última configuración que utiliza 8 bits. Los valores correspondientes para cada permutación *t* se encuentran en la Tabla I. La cantidad de rondas N_r siempre es de 12 vueltas para todo valor de *t*. La celda del estado interno se denota con la fila *i* y la columna *j* siendo *S[i,j]* con valores $0 \leq i, j < d$.

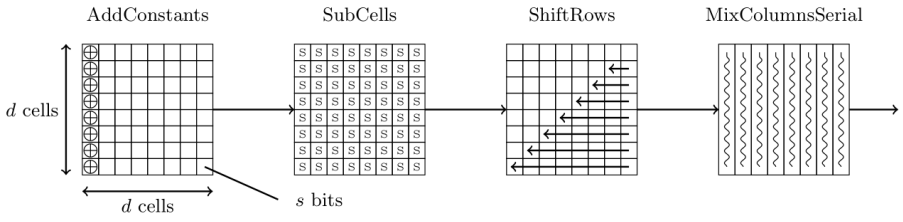


Figura 4: Esquema permutaciones en una ronda de Photon. (Guo et. al., 2011)

Tabla I. Parámetros de las permutaciones internas P_t con las constantes internas IC_d, el polinomio irreducible, y los coeficientes Z_i . (Guo et. al., 2011).

	t	D	s	N_r	IC_d	pol. Irr.	coef. Z_i
P_{100}	100	5	4	12	[0, 1, 3, 6, 4]	x^4+x+1	(1, 2, 9, 9, 2)
P_{144}	144	6	4	12	[0, 1, 3, 7, 6, 4]	x^4+x+1	(1, 2, 8, 5, 8, 2
P_{196}	196	7	4	12	[0, 1, 2, 5, 3, 6, 4]	x^4+x+1	(1, 4, 6, 1, 1, 6, 4)
P_{256}	256	8	4	12	[0, 1, 3, 7, 15, 14, 12, 8]	x^4+x+1	(2, 4, 2, 11, 2, 8, 5, 6)
P_{288}	288	6	8	12	[0, 1, 3, 7, 6, 4]	$x^8+x^4+x^3+x+1$	(2, 3, 1, 2, 1, 4)

Las capas se detallan a continuación:

AddConstant: Se utiliza un arreglo RC con constantes a utilizar en cada ronda, el cual ha sido generado de antemano con la idea de ahorrar tiempo de cálculo en hardware, forzando que cada computación de la ronda sea diferente. Las mismas pueden ser generadas con corrimiento de bits lineales o LFSR de muy alta performance, siendo *RC(v) = [1,3,7,14,13,11,6,12,9,2,5,10]* donde *v* es el número de ronda contando desde 1. Se aplica la operación XOR al RC(v) correspondiente con cada celda de la primer columna del estado interno. Luego se aplica otra operación XOR, esta vez con las constantes internas y a la misma primera columna del estado interno. En resumen:

$$S'[i,0] = S'[i,0] \oplus RC(v) \oplus IC_d(i) \quad \text{con } 0 \leq i < d \tag{16}$$

SubCells: Aplica una S-box de *s* bits a cada celda del estado interno. Se utilizaron S-boxes de 4 bits de tamaño con el fin de ser lo más compacta posible en hardware, donde tamaños de 4 u 8 facilitan la implementación en software.

$$S'[i,j] = SBOX(S[i,j]) \text{con } 0 \leq i,j < d \tag{17}$$

ShiftRows: Al igual que en AES, para cada fila *i* se rotan todas las celdas hacia la izquierda y por la cantidad de columnas *i*, siendo el estado interno siempre una matriz cuadrada, donde la fila *i* se rota *i* posiciones a la izquierda comenzando por 0.

$$S'[i,j] = S'[i,(j+i) \bmod d] \text{ con } 0 \leq i,j < d \tag{18}$$

MixColumnsSerial: Está basada en la homónima función implementada en AES, aunque debió ser modificada para mejorar el espacio requerido al realizar los cómputos.

La matriz se construye comenzando con un vector con un 1 en la última posición, y corriendo el 1 hacia la izquierda en los sucesivos vectores que conforman la matriz desde abajo hacia arriba. Los coeficientes Z_0 a Z_{d-1} son elegidos libremente, como en la Figura 5. Luego, se computa la potencia de la matriz, multiplicándola por ella misma *d* veces, como en la Figura 6.

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix}$$

Figura 5: Ejemplo de matriz A (d x d), cuya última fila es elegida libremente. (Guo et. al., 2011)

$$(A)^4 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 1 & 4 \end{pmatrix}^4 = \begin{pmatrix} 1 & 2 & 1 & 4 \\ 4 & 9 & 6 & 17 \\ 17 & 38 & 24 & 66 \\ 66 & 149 & 100 & 11 \end{pmatrix}$$

Figura 6: Ejemplo del cálculo de una matriz predefinida en Photon. (Guo et. al., 2011)

Esta última capa es aplicada a cada columna del estado interno independientemente.

$$(S\,0\,[0,j],\ldots,S\,0\,[d\,-\,1,j])^{\,T} \;=\; A_t^d \times\; (S[0,j],\ldots,S[d\,-\,1,j])^{\,T} \qquad (19)$$

$$\text{siendo } A_t \;=\; Serial(Z_0,\ldots,Z_{d-1}\,) \text{ con } 0 \;\leq\; j \;<\; d \qquad (20)$$

4.2. Quark

Quark (Aumasson et al. 2010) es el segundo algoritmo de criptografía liviana para función hash seleccionado, nombrado a partir de la partícula subatómica en clara alusión a lo compacto de la función. Toma prestado decisiones de diseño aplicadas en otros algoritmos como el algoritmo de cifrado de flujo Grain (Hell, Johansson y Meier 2007), y para el algoritmo de cifrado de bloque Katan (Cannière, Dunkelman y Knežević 2009).

De igual manera, se especifican las constantes y funciones utilizadas para este algoritmo en su configuración más liviana U-QUARK. La implementación en C# realizada es una adaptación del código fuente original (Aumasson 2020).

4.2.1. Constantes

MAXDIGEST: Tamaño del digesto en unidades de *nibble*. Para la configuración básica, el valor de 48 *nibbles* representa 12 caracteres hexadecimales.

RATE: Longitud del bloque o tasa de bytes con la cual se realizarán las permutaciones y relleno. Valor: 1.

WIDTH: Tamaño interno de la esponja en bytes. Valor: 17.

DIGEST: Tamaño en bytes definido con el mismo tamaño establecido en WIDTH.

IV: Vector interno predefinido con 17 valores hexadecimales. Los valores elegidos se tomaron a partir de aplicar la función SHA-256 al nombre de la implementación de cada variante de Quark. Para la versión mínima, denominada U-QUARK se tomaron los primeros valores hexadecimales del digesto obtenido en la función SHA256("u-quark").

ROUNDS_U: Cantidad de rondas. Valor 4 x 136 para u-quark.

N_LEN_U: Longitud de *n*. Valor 68 para u-quark.

L_LEN_U: Longitud de *l.* Valor 10 para u-quark.

Como referencia, los valores en bits de los parámetros utilizados en la configuración mínima son: Rate (*r*): 8; Capacity (*c*): 128; Width (*b*): 136; Rounds (*4b*): 544; Digest (*n*): 136.

4.2.2. Funciones

Se comienza definiendo una estructura denominada HashState que contiene una variable entera *pos* con el número de bytes leídos en el bloque procesado, y un arreglo de valores enteros (sin signo) *x* de posiciones iguales a *WIDTH x 8*. Esta estructura contendrá el estado de ejecución de la función hash, con 136 posiciones para la configuración mínima.

Init: Se procede con la inicialización del HashState *state* con una operación de corrimiento de bits aplicada al *IV* o Vector de Inicialización. En esta etapa, se rellena el mensaje con un bit de valor 1, seguido de tantos bits 0 como sea posible, hasta alcanzar una longitud múltiplo de RATE, es decir la longitud de bloque.

En la fase de absorción, se le aplica una operación XOR a los bloques del mensaje en conjunto con la cantidad definida en RATE de bits del estado *x,* aplicando operaciones de permutación.

Update: Se transforma el mensaje de entrada en un vector conteniendo un carácter por posición. El tamaño del vector será idéntico a la cantidad de caracteres que posea la cadena de entrada. Para cada bit, se aplica una operación XOR, y una permutación cada vez que la posición del estado coincida con la constante RATE.

Permute: La operación de permutación consiste en definir 2 arreglos de enteros sin signo NFSR o de retroalimentación no lineal y de tamaño definido por la suma de las constantes N_LEN_U y ROUNDS_U denominados *xx (NFSR X)* e *yy (NSFR Y)*, y L_LEN_U y ROUNDS_U para el vector LFSR o *Linear Feedback Shift Register,* vector cuyo estado se actualiza a partir de su propio estado previo (ver Figura 7). La función recibe el estado *x* de la estructura HashState. Inicializa los vectores mediante dos iteraciones *for* secuenciales, copiando la primer mitad del estado *x* en *xx*, y la segunda mitad en *yy,* e inicializando el vector LFSR con una secuencia de 1. Mediante un tercer bucle *for,* realiza todas las operaciones relacionadas con las funciones *f* y *g* para computar un valor entero sin signo denominad *h,* el cual es retroalimentado en una operación XOR a los vectores *xx* e *yy* nuevamente. Finalmente, en un bucle *for* final, se copia el estado final a *x*.

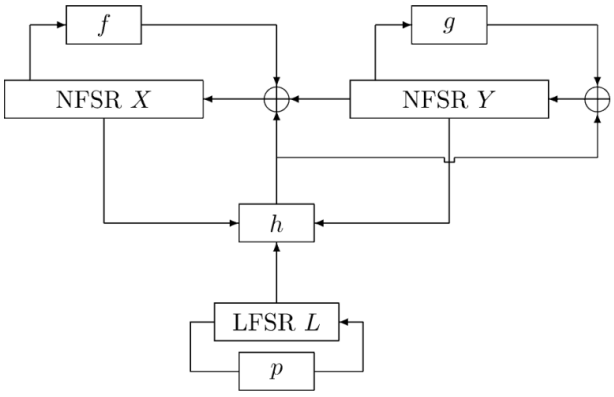


Figura 7: Diagrama de permutación empleado en Quark.(Aumasson et al. 2010)

Las funciones denotadas como *f, g* y *h* realizan operaciones AND y XOR entre distintas posiciones predefinidas de los vectores *xx (NFSR X)* e *yy (NSFR Y),* como se detalla a continuación (para la configuración mínima de Quark, y un registro de 68 bits):

Función f:

$$X_0 + X_9 + X_{14} + X_{21} + X_{28} + X_{33} + X_{37} + X_{45} + X_{50} + X_{52} + X_{55}$$
$$+X_{55}\,X_{59} + X_{33}X_{37} + X_9X_{15} + X_{45}\,X_{52}X_{55} + X_{21}\,X_{28}X_{33}$$
$$+X_9X_{28}X_{45}X_{59} + X_{33}X_{37}X_{52}X_{55} + X_{15}\,X_{21}X_{55}X_{59}$$
$$+X_{37}X_{45}X_{52}X_{55}X_{59} + X_9X_{15}X_{21}X_{28}X_{33} + X_{21}X_{28}X_{33}X_{37}X_{45}X_{52}$$

Función g:

$$Y_0 + Y_7 + Y_{16} + Y_{20} + Y_{30} + Y_{35} + Y_{37} + Y_{42} + Y_{49} + Y_{51} + Y_{54}$$
$$+Y_{54}\,Y_{58} + Y_{35}Y_{37} + Y_7Y_{15} + Y_{42}Y_{51}Y_{54} + Y_{20}Y_{30}Y_{35}$$
$$+Y_7Y_{30}Y_{42}Y_{58} + Y_{35}Y_{37}Y_{51}Y_{54} + Y_{15}Y_{20}Y_{54}Y_{58}$$
$$+Y_{37}Y_{42}Y_{51}Y_{54}Y_{58} + Y_7Y_{15}Y_{20}Y_{30}Y_{35} + Y_{20}Y_{30}Y_{35}Y_{37}Y_{42}Y_{51}$$

Función h:

$$L_0 + X_1 + Y_2 + X_4 + Y_{10} + X_{25} + X_{31} + Y_{43} + X_{56} + Y_{59}$$
$$+Y_3X_{55} + X_{46}X_{55} + X_{55}Y_{59} + Y_3X_{25}X_{46} + Y_3X_{46}X_{55}$$
$$+Y_3X_{46}Y_{59} + L_0X_{25}X_{46}Y_{59} + L_0X_{25}$$

En la fase de estrujamiento, se devuelven la cantidad de RATE bits en las últimas posiciones del estado *x,* aplicando operaciones de permutación, hasta que se alcance la cantidad definida en la constante MAXDIGEST.

Final: Realiza una operación XOR con el bit 1 en determinadas posiciones del estado *x*. Realiza una última operación de permutación. Inicializa el vector de salida con 0. Mediante un bucle *while* y *for* anidado, realiza una operación que involucra las operaciones XOR y corrimiento de bits a la izquierda para obtener un byte de salida a la vez. Y se realiza una permutación extra cada vez que se extraigan la cantidad de bytes establecidos en la constante RATE.

4.3.Spongent

El tercer algoritmo seleccionado a evaluar es Spongent (Bogdanov et al. 2011), el cual también está basado en el algoritmo esponja. Toma ideas de diseño del algoritmo de cifrado de bloques Present. La configuración más liviana a ser evaluada será Spongent-88/80/8, mediante

una adaptación del código fuente optimizado para un microcontrolador ATtiny45 que asegura haber optimizado el código para reducir el impacto en el tamaño que ocupa (Groot 2019).

4.3.1. Constantes

b: Suma resultante entre tasa de bits *r* y capacidad *c*. Valor: 88.

B: Es *b* expresado en bytes, es decir *b / 8*.

n: Bits de salida del digesto. Valor: 88.

State: Vector estado de 34 posiciones.

S: Vector S-Box de 16 posiciones conteniendo valores hexadecimales predefinidos.

Basado en una permutación de *b* cantidad de bits llamada π_b con capacidad de *c* bits y una tasa de bits de *r*. Los mensajes m_i tienen longitud *r y* los valores h_i son parte del digesto obtenido (ver Figura 8). Se trata de obtener una salida de longitud fija mediante una entrada variable, aplicando sucesivas permutaciones en un vector estado de longitud fija de *b* bits. El estado interno está definido por $b = r + c \geq n$ es la amplitud o WIDTH.

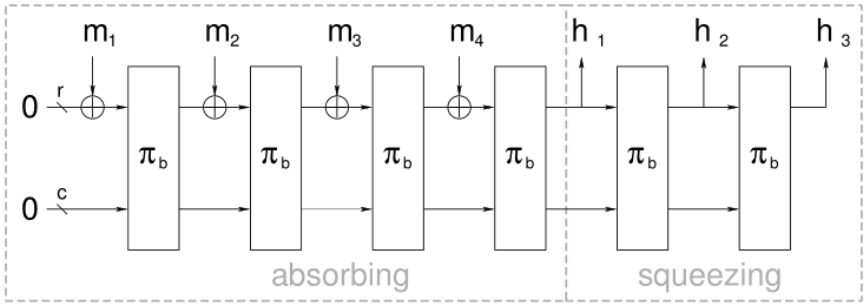


Figura 8: Diagrama del algoritmo esponja empleado en Spongent. (Bogdanov et al. 2011)

4.3.2. Funciones

Init: La fase inicial consiste en rellenar el mensaje con un bit 1 y tantos bits 0 como sean necesarios hasta completar un múltiplo de *r*.

Absorbing: Se aplica una operación XOR entre los primeros *r* bits del mensaje de entrada junto con los primeros *r* bits del estado, llevando a cabo operaciones de permutación entre medio.

Squeezing: Los primeros *r* bits del estado se devuelven como salida, previa aplicación de permutaciones, hasta que se alcancen los *n* bits de longitud del digesto de salida.

Las permutaciones se realizan a partir del algoritmo Present, en un ciclo FOR de cantidad de vueltas *R*, el cual depende del tamaño de bloque *b* (ver Figura 9).

for $i = 1$ to R **do**
  STATE ← $\text{lCounter}_{(\text{inv})b}(i) \oplus \text{STATE} \oplus \text{lCounter}_b(i)$
  STATE ← $\text{sBoxLayer}_b(\text{STATE})$
  STATE ← $\text{pLayer}_b(\text{STATE})$
end for

Figura 9: Permutación al estilo Present en Spongent. (Bogdanov et al. 2011)

lCounter es el estado del vector LFSR dependiente de *b* en el momento *i*. Sus bits son movidos cada vez que el estado haya sido utilizado y el valor de su estado final sean todos 1.

$\text{lCounter}_{(\text{inv})}$ representa el estado invertido del vector homónimo.

sBoxLayer denota el uso del vector S-Box preconfigurado (ver Figura 10).

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

Figura 10: Vector S-Box empleado en Spongent. (Bogdanov et al. 2011)

pLayer es una extensión de la permutación empleada en Present pero invertida (ver Figura 11).

$$P_b(j) = \begin{cases} j \cdot b/4 \mod b - 1, & \text{if } j \in \{0, \ldots, b-2\} \\ b-1, & \text{if } j = b-1. \end{cases}$$

Figura 11: Función pLayer aplicada en Spongent. (Bogdanov et al. 2011)

5. Resultados

Finalizada la implementación de las tres funciones hash, se procede a la programación de un programa conductor que registre los tiempos de ejecución de dichas funciones, denominado *Profiler*.

Se crean cinco vectores de 32 posiciones cada uno. Cada posición representa una longitud de cadena de caracteres de entrada o *mensaje* que deberá procesar cada función. Se evalúan distintos rangos de longitud: El primer rango será de 0 a 9 caracteres, el segundo será de 10 a 100 caracteres (de 10 en 10), el tercer rango será de 200 a 1000 caracteres (de 100 en 100), y finalmente el cuarto rango será solamente de 10.000, 100.000 y 1.000.000 caracteres de entrada.

Además del valor constante predefinido POS de 32, se define otro valor CYCLES de 1.000 unidades que representa la cantidad de veces que se llamará a la función hash para que procese un mismo mensaje. De esta manera, se intenta encontrar un promedio de tiempo de ejecución para una misma longitud de entrada, compensando las variaciones que pudieran existir al compartir el uso del procesador con otras aplicaciones o servicios. El valor de CYCLES podrá ajustarse para las longitudes de entrada más demandantes.

El primer vector se inicializa con cadenas de caracteres obtenidas al azar de distintas longitudes especificadas anteriormente, incluida la cadena vacía de longitud cero. Los otros cuatro vectores contendrán en cada posición los resultados de las mediciones efectuadas para cada cantidad de longitud de entrada predefinida. En *totalTime* se almacena el tiempo total empleado al ejecutar la función hash tantos ciclos como fue predefinido. En *oneTime* se registra el tiempo de ejecución que tomó procesar solamente una cadena de entrada. En *avgTime* se registra el tiempo promedio de ejecución por mensaje de entrada. Finalmente, en *hashRate* se registra el valor teórico de hashes por segundo alcanzado.

Se definen dos variables *startTime* y *endTime* para registrar el tiempo de inicio y fin de la ejecución de tantos ciclos predefinidos. El tiempo se registra utilizando el reloj interno, expresado en UTC (Tiempo Coordinado Universal). El valor de inicio se obtiene inmediatamente antes del ciclo *for* que iterará los sucesivos ciclos. Asimismo, el valor de finalización se registra inmediatamente después de la culminación de todos los ciclos de ejecución.

Completados los ciclos de ejecución, se define *interval* como la variable a emplear en el cálculo del tiemplo empleado para obtener las funciones hash, realizando:

$$interval = endTime - startTime \quad (21)$$

También se registran en cada posición de los vectores mencionados: el tiempo total empleado en unidades de segundo, el tiempo promedio por ciclo, y la tasa de hash por segundo.

$$totalTime[i] = interval.TotalSeconds \quad (22)$$

$$avgTime[i] = \frac{interval.TotalSeconds}{CYCLES} \quad (23)$$

$$hashRate[i] = \frac{CYCLES}{interval.TotalSeconds} \quad (24)$$

Antes de finalizar un ciclo, se calcula un hash individual para registrar los tiempos de ejecución individual y compararlo con el valor promedio. Cabe destacar que los digestos obtenidos son desestimados para esta evaluación.

El programa conductor posee funciones auxiliares para llevar a cabo el proceso: Una función *GetARandomString* que recibe un número entero y devuelve una cadena de caracteres al azar con la longitud especificada con el parámetro de entrada; una función *GetAllRandomStrings* que hace uso de la función anterior y devuelve un vector de cadenas de caracteres al azar en un proceso que involucra cuatro ciclos *for* secuenciales que sirven para ajustar el parámetro de entrada de la longitud de la cadena de caracteres al azar entre los rangos definidos.

Para obtener una secuencia de caracteres al azar, se utiliza la librería definida en *System.Random,* definiendo una variable estática inicializada una sola vez que permite mantener el estado entre sucesivas llamadas para garantizar que la cadena devuelta sea distinta, ya que al utilizar como semilla el estado del reloj interno y en sucesivos llamados consecutivos podría pasar que el estado del reloj no cambiase.

Las cadenas de caracteres al azar se generan y almacenan en un vector antes de comenzar con la evaluación del algoritmo. Se eligen cadenas al azar para considerar que dos cadenas de caracteres de igual longitud podrían diferir en su tiempo de procesamiento por tener que aplicar un relleno a algunas. En Photon, las funciones de rellenado solo dependen de la longitud de la cadena.

En sucesivas pruebas, la primera llamada a la función hash puede verse afectada por la compilación JIT o “Just In Time compiler” (Warren 2021b) que convierte el código a código nativo de cada máquina a demanda en el instante en que se ejecuta. Se toma en cuenta que JIT puede no compilar el código de la función hasta el primer llamado para ahorrar tiempo. Esto se

vio comprobado al realizar pruebas, donde el primer llamado a la función demoraba aproximadamente 0,5 milisegundos extra que los llamados siguientes.

5.1 Framework de pruebas en procesador de alto rendimiento

La aplicación se desplegó con el modo "autocontenido" para generar un ejecutable en vez de un módulo dependiente del entorno, y utilizando el framework *netcoreapp2.1 win-x64.* Para ejecutarlo, se lo instancia con el comando *START* y el parámetro /REALTIME para indicar al sistema operativo la necesidad de obtener la máxima prioridad para utilizar los recursos del sistema, además de ajustar el modo de ahorro de energía en la configuración de máxima performance. La configuración del sistema para realizar las pruebas es: Sistema operativo Windows 10, procesador Intel Core i7-9750H con 6 núcleos y 12 hilos a 2,60 GHz, y memoria RAM SODIMM DDR4 2667 MHz (0,4 ns).

5.1.1 Tiempo de ejecución

Los resultados obtenidos para el algoritmo empleado en la función hash Photon se visualizan en la Tabla II. Arrojan un tiempo promedio de ejecución cercano a los 0,2 ms y una tasa de procesamiento cercana a los 5.000 hashes/segundo para cadenas de entrada vacías y de hasta 2 caracteres. El tiempo promedio de ejecución se mantiene por debajo de 1 ms para entradas menores a 50 caracteres, con una tasa de ejecución cercana a 1.000 hashes/s. El tiempo promedio asciende a 10 ms para entradas de 700 caracteres, donde la tasa es inferior a los 100 hashes/s. Finalmente, para las entradas más demandantes: la de 10.000 caracteres alcanza una tasa de trabajo de 3,14 hashes/s, una sola cadena de 100.000 se procesa en 3 segundos, mientras que una de 1.000.000 de caracteres demanda 31,25 segundos. El hash por unidad de tiempo se obtuvo al procesar cadenas de 30.000 caracteres, realizando una extrapolación de los datos de la tabla.

Tabla II. Registro de valores temporales y tasa de hashes para el algoritmo Photon en equipo de alto rendimiento.

No.	Longitud Cadena de Entrada [carác.]	Tiempo ejecución 1 entrada [s]	Tiempo ejecución 1 ciclo [s]	Tiempo promedio ejec. por entrada [s]	Tasa de Hashes [h/s]
0	0	0,0001982	0,2099792	0,00020998	4762,376
1	1	0,0001928	0,1993273	0,00019933	5016,874

2	2	0,0001967	0,1991999	0,00019920	5020,083
3	3	0,0002351	0,2396402	0,00023964	4172,923
4	4	0,0002403	0,2431818	0,00024318	4112,150
5	5	0,0002809	0,2805027	0,00028050	3565,028
6	6	0,0002810	0,2895022	0,00028950	3454,205
7	7	0,0002777	0,2841266	0,00028413	3519,558
8	8	0,0003169	0,3220094	0,00032201	3105,499
9	9	0,0003147	0,3222117	0,00032221	3103,550
10	10	0,0004391	0,3620226	0,00036202	2762,258
11	20	0,0005142	0,5212507	0,00052125	1918,463
12	30	0,0006999	0,6876660	0,00068767	1454,194
13	40	0,0008334	0,8464177	0,00084642	1181,450
14	50	0,0009903	1,0050887	0,00100509	994,937
15	60	0,0011501	1,1720929	0,00117209	853,175
16	70	0,0013106	1,3309001	0,00133090	751,371
17	80	0,0014648	1,4937633	0,00149376	669,450
18	90	0,0016233	1,6552245	0,00165522	604,148
19	100	0,0017826	1,8218989	0,00182190	548,878
20	200	0,0036224	3,4331585	0,00343316	291,277
21	300	0,0050931	5,0475986	0,00504760	198,114
22	400	0,0064916	6,6099661	0,00660997	151,287
23	500	0,0082068	8,2547047	0,00825470	121,143
24	600	0,0098426	9,8619519	0,00986195	101,400
25	700	0,0112299	11,4320189	0,01143202	87,474
26	800	0,0130627	13,0182734	0,01301827	76,815
27	900	0,0151858	14,6976432	0,01469764	68,038
28	1.000	0,0162123	16,3192237	0,01631922	61,277
29	10.000	0,1551309	31,8333598	0,31833350	3,141
30	100.000	1,5520480	313,6504868	3,13650482	0,319
31	1.000.000	15,6512826	3125,883078	31,25883070	0,032

Utilizando idéntica configuración y metodología, los resultados para la función hash Quark se presentan en la Tabla III. Aparecen valores cercanos a 0,4 ms y una tasa de procesamiento cercana a los 2.500 hashes/segundo para cadenas de 0 a 3 caracteres de longitud. El tiempo promedio de ejecución supera mínimamente 1 ms al llegar a los 40 caracteres de entrada, mientras que la tasa de ejecución se coloca por debajo de los 1.000 hashes/s. Los 10 ms de tiempo de ejecución se alcanzan al procesar 500 caracteres de entrada, con una tasa de hashes procesados inferior a los 100 hashes/s. Para las entradas más pesadas: 10.000 caracteres procesados en 0,4 segundos, 100.000 en 4,13 segundos, y 1.000.000 en 40,79 segundos. Este algoritmo alcanzó la tasa de procesamiento del hash/segundo con una cadena de caracteres aproximada de 25.000.

Tabla III. Registro de valores temporales y tasa de hashes para el algoritmo Quark en equipo de alto rendimiento.

No.	Longitud Cadena de Entrada [carác.]	Tiempo ejecución 1 entrada [s]	Tiempo ejecución 1 ciclo [s]	Tiempo promedio ejec. por entrada [s]	Tasa de Hashes [h/s]
0	0	0,0003475	0,3726299	0,00037263	2683,628
1	1	0,0003690	0,3705518	0,00037055	2698,678
2	2	0,0003801	0,3896421	0,00038964	2566,458
3	3	0,0003971	0,4070653	0,00040707	2456,608
4	4	0,0004167	0,4353200	0,00043532	2297,161
5	5	0,0004387	0,4498715	0,00044987	2222,857
6	6	0,0004901	0,4708553	0,00047086	2123,795
7	7	0,0004779	0,4885181	0,00048852	2047,007
8	8	0,0005108	0,5081970	0,00050820	1967,741
9	9	0,0006991	0,5291065	0,00052911	1889,979
10	10	0,0005366	0,5493453	0,00054935	1820,349
11	20	0,0007362	0,7505786	0,00075058	1332,306
12	30	0,0009343	0,9517157	0,00095172	1050,734
13	40	0,0011114	1,1517173	0,00115172	868,269
14	50	0,0013317	1,3553537	0,00135535	737,815
15	60	0,0015306	1,5594650	0,00155947	641,246
16	70	0,0017284	1,7652756	0,00176528	566,484
17	80	0,0019498	1,9570552	0,00195706	510,972
18	90	0,0022427	2,1813192	0,00218132	458,438
19	100	0,0023779	2,3909590	0,00239096	418,242
20	200	0,0043254	4,4257410	0,00442574	225,951
21	300	0,0067056	6,4449842	0,00644498	155,159
22	400	0,0083853	8,4434467	0,00844345	118,435
23	500	0,0105126	10,4743565	0,01047436	95,471
24	600	0,0124923	12,5087343	0,01250873	79,944
25	700	0,0144284	14,5363771	0,01453638	68,793
26	800	0,0160963	16,5139446	0,01651394	60,555
27	900	0,0193458	18,6124595	0,01861246	53,727
28	1.000	0,0202530	22,2975287	0,02229753	44,848
29	10.000	0,2069978	40,6906136	0,40690610	2,458
30	100.000	2,0635055	413,9824531	4,13982450	0,242
31	1.000.000	20,3864328	4079,8484154	40,79848410	0,025

Para Spongent, los resultados se presentan en la Tabla IV. Se toman ciclos de 100 pruebas por longitud de cadena de entrada. No está implementado el digesto de una cadena vacía, por

lo tanto no se lo considera en el análisis. Los tiempos de ejecución abarcan de 1 ms a 2 ms para cadenas de menos de 10 caracteres. La tasa de hashes por segundo es inferior a los 1.000 hashes/s con solo un carácter de entrada. Los 10 ms de tiempo de procesamiento se alcanzan al superar los 100 caracteres de entrada. El hash/s se alcanza al procesar 13.000 caracteres de entrada, aproximadamente.

Tabla IV. Registro de valores temporales y tasa de hashes para el algoritmo Spongent en equipo de alto rendimiento.

No.	Longitud Cadena de Entrada [carác.]	Tiempo ejecucion 1 entrada [s]	Tiempo ejecucion 1 ciclo [s]	Tiempo promedio ejec. por entrada [s]	Tasa de Hashes / s
0	1	0.0010803	0.1144141	0.001144141	874.018150
1	2	0.0011794	0.1251172	0.001251172	799.250623
2	3	0.0012145	0.1323344	0.001323344	755.661415
3	4	0.0013559	0.1418323	0.001418323	705.058016
4	5	0.0013943	0.1444484	0.001444484	692.288734
5	6	0.0015058	0.1508975	0.001508975	662.701503
6	7	0.0014711	0.1501574	0.001501574	665.967844
7	8	0.0016667	0.1662474	0.001662474	601.513167
8	9	0.0016238	0.1688995	0.001688995	592.068064
9	10	0.0018840	0.1860004	0.001860004	537.633252
10	20	0.0025561	0.2586725	0.002586725	386.589220
11	30	0.0031700	0.3310461	0.003310461	302.072732
12	40	0.0041128	0.4381265	0.004381265	228.244582
13	50	0.0049047	0.4774896	0.004774896	209.428645
14	60	0.0060783	0.5706541	0.005706541	175.237504
15	70	0.0065304	0.6500065	0.006500065	153.844615
16	80	0.0070826	0.7122773	0.007122773	140.394759
17	90	0.0077803	0.7780373	0.007780373	128.528542
18	100	0.0086970	0.8592046	0.008592046	116.386714
19	200	0.0156763	1.6120523	0.016120523	62.032727
20	300	0.0227980	2.3603278	0.023603278	42.366997
21	400	0.0305080	3.1637350	0.031637350	31.608210
22	500	0.0383690	3.8759105	0.038759105	25.800389
23	600	0.0488801	5.1422059	0.051422059	19.446907
24	700	0.0544621	5.5512460	0.055512460	18.013974
25	800	0.0619844	6.2732808	0.062732808	15.940622
26	900	0.0694714	6.8167241	0.068167241	14.669803
27	1.000	0.0768953	7.5997121	0.075997121	13.158393
28	10.000	0.7614124	76.9713447	0.769713447	1.299185
29	100.000	13.6718012	277.5160264	2.775160262	0.036034
30	1.000.000	138.1418962	2748.7252314	27.487935911	0.003638

En líneas generales, Photon demuestra ser más veloz con respecto a Quark y Spongent a igual longitud de mensaje de entrada.

Se detecta una dispersión en los tiempos de ejecución registrados para cadena de caracteres de longitud menor a 10, y un tiempo de procesamiento notoriamente mayor para la cadena vacía. Esto puede suceder por el tiempo de respuesta retrasado que puede provocar la primera llamada a la función, como se explicó anteriormente debido a la compilación JIT. Podría omitirse el cálculo de un digesto para una cadena vacía, siempre que exista un resultado predefinido. En el caso de Spongent, no arroja resultados para mensajes vacíos.

Se evidencia una relación lineal entre los tiempos de procesamiento y la longitud de la cadena de entrada para ambos algoritmos, donde los valores se estabilizan sobre la línea de tendencia para cadena de caracteres más amplias (ver Figuras 12, 13 y 14).

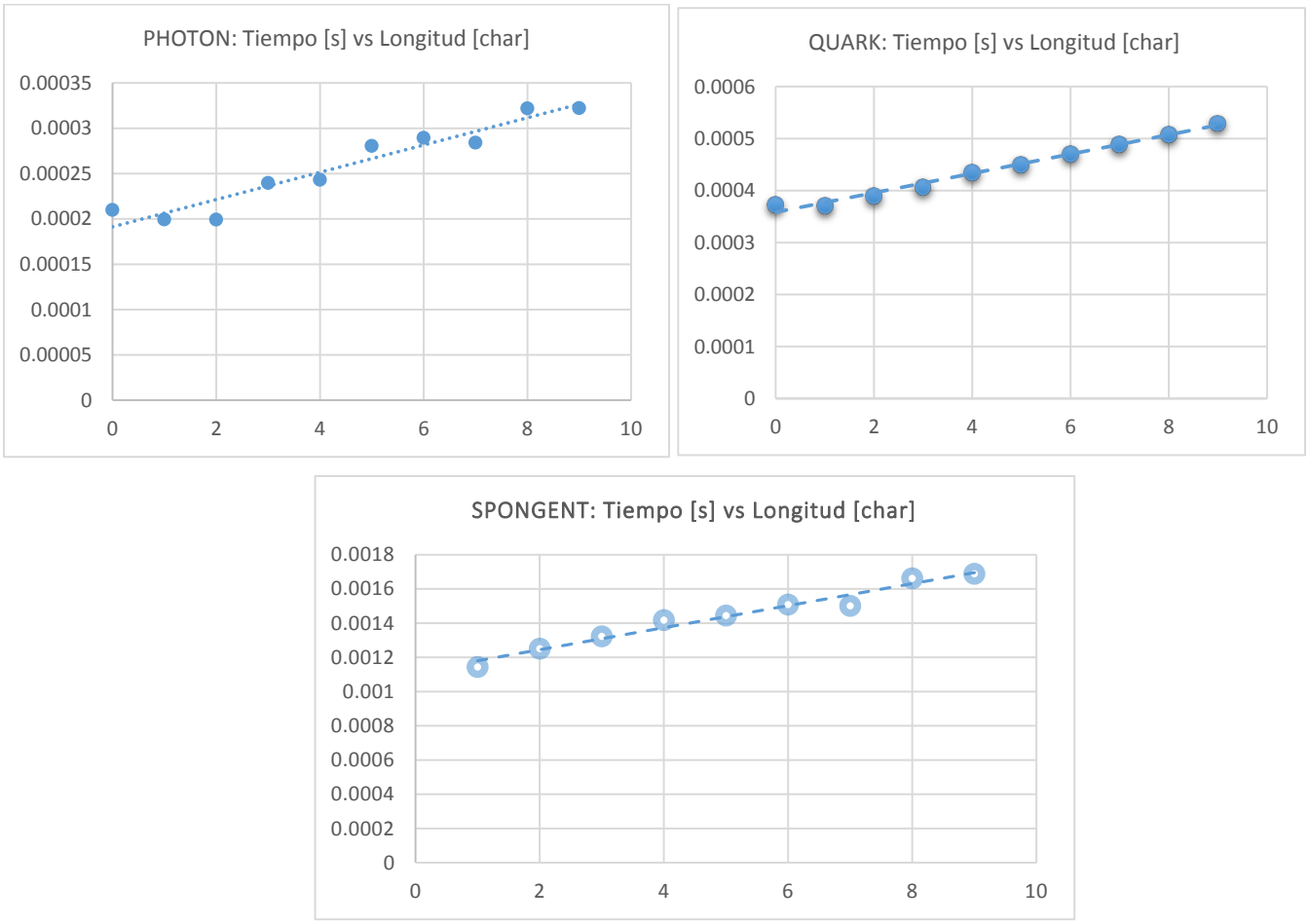


Figura 12. Gráficos de dispersión lineal para Photon, Quark y Spongent. Tiempo [s] vs. Longitud [carac.] para longitud de entrada entre 0 y 9 caracteres.

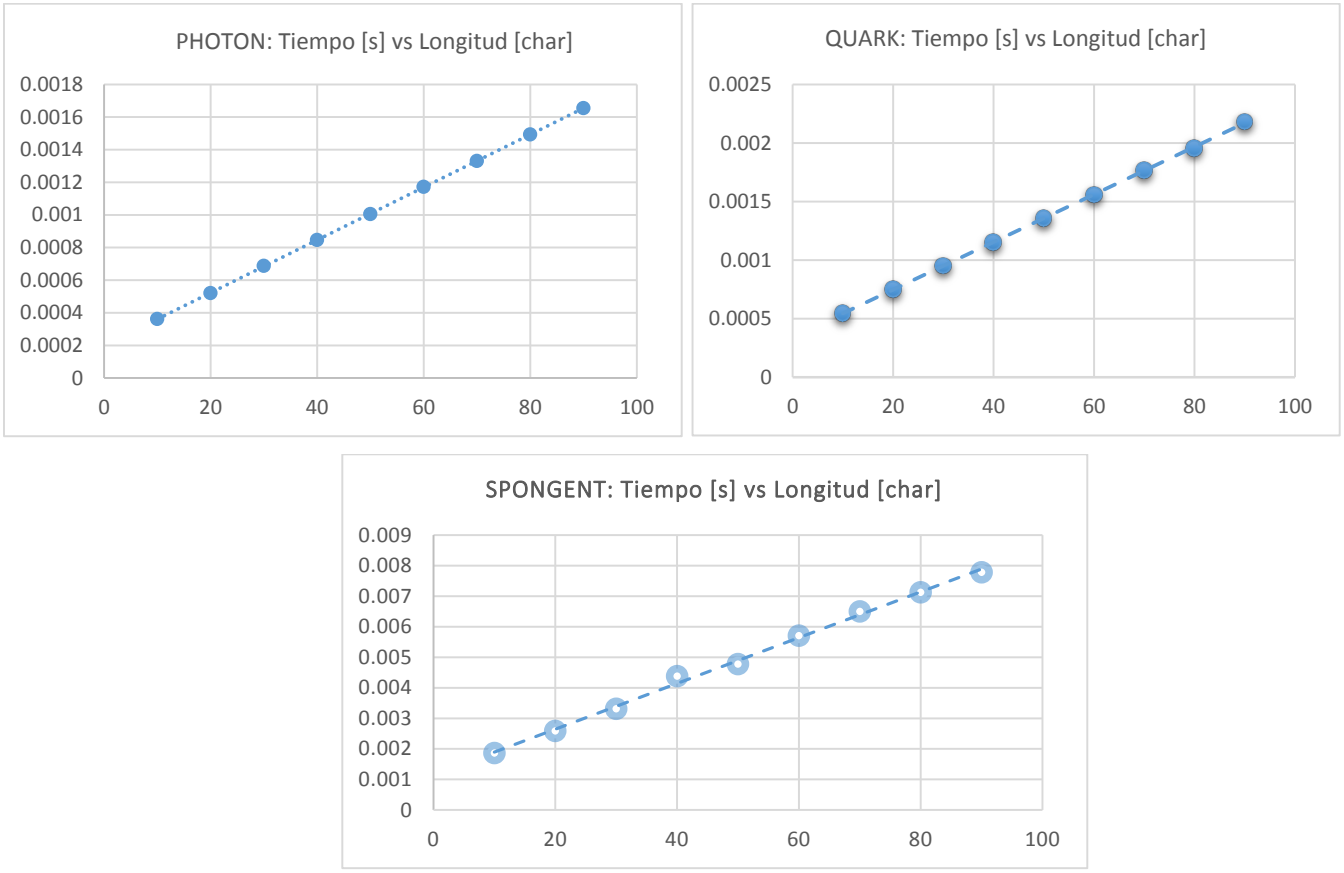
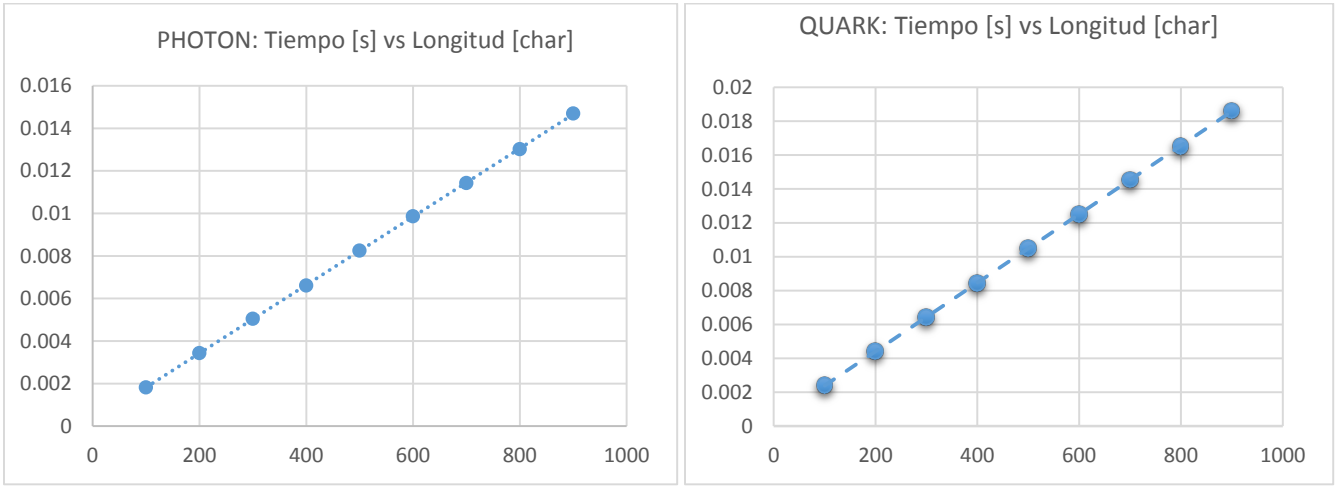


Figura 13. Gráficos de dispersión lineal para Photon, Quark y Spongent. Tiempo [s] vs. Longitud [carác.] para longitud de entrada entre 10 y 90 caracteres.



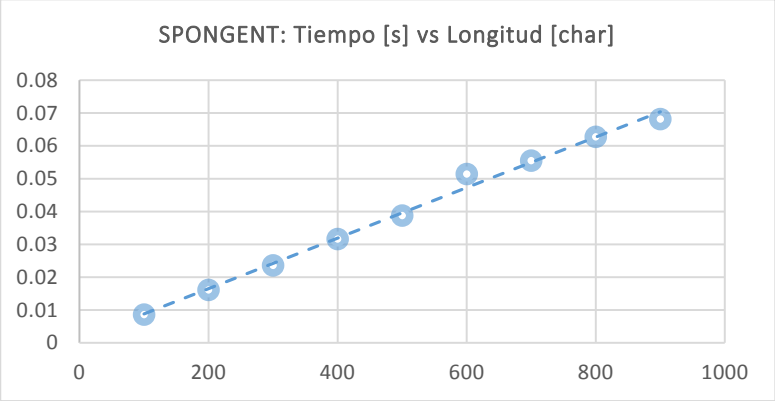


Figura 14. Gráficos de dispersión lineal para Photon, Quark y Spongent. Tiempo [s] vs. Longitud [carác.] para longitud de entrada entre 100 y 900 caracteres.

5.1.2 Tamaño de código

Observando la cantidad de líneas de código se puede obtener un panorama de la cantidad de instrucciones que realiza cada algoritmo. En una primera revisión, la parte relevante del algoritmo, es decir su función principal y sus funciones auxiliares, Photon supera a Quark en complejidad: 219 líneas para Photon versus 169 líneas para Quark. No obstante, Spongent lidera con solo 102 líneas. Visualizando la cantidad de instrucciones generadas en *Intermediate Language* (Warren 2021c), el lenguaje de bajo nivel producido post compilación que el *Common Language Runtime* o *CLR* (Warren 2021a) ejecutará, se aprecian un poco más del 50% por encima de la cantidad de instrucciones generadas para Photon en el algoritmo Quark: 785 para Photon versus 1225 para Quark, mientras que Spongent vuelve a liderar con solo 445 instrucciones. La herramienta empleada para extraer y visualizar las instrucciones IL es LINQPad 5 («LINQPad - The .NET Programmer's Playground» 2021). Visualizar las instrucciones IL revela la verdadera complejidad del algoritmo Quark, donde la mayor cantidad de instrucciones yacen en la función de permutación con 950 instrucciones, involucrando una secuencia de operaciones XOR y AND reiteradas veces en el cálculo de las funciones *f, g* y *h* (ver Figura 15).

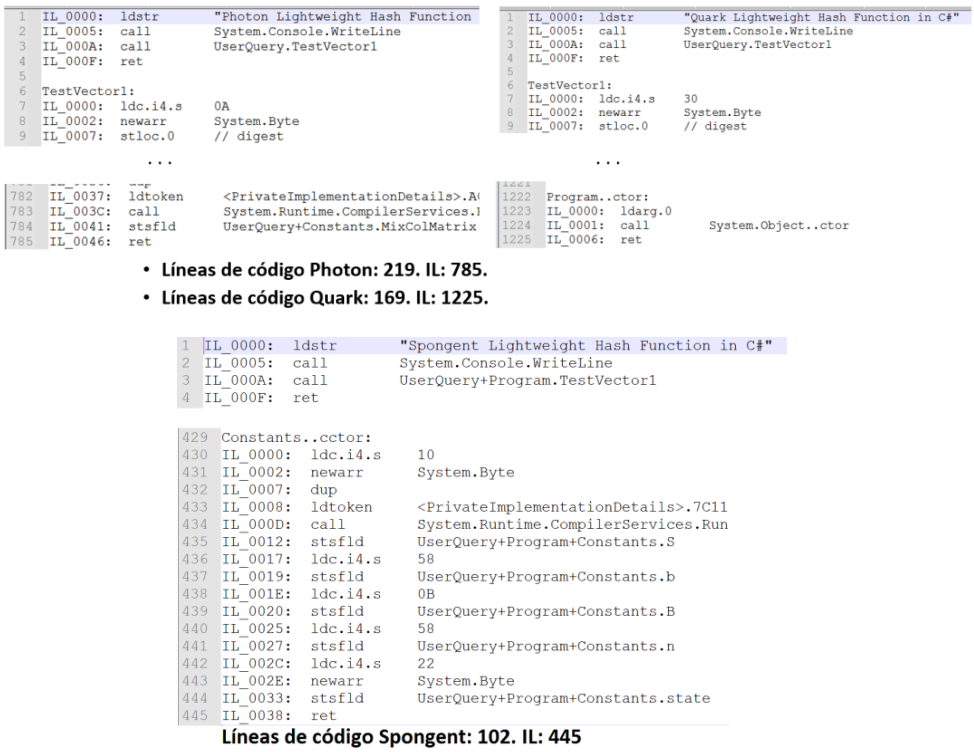


Figura 15. Cantidad de líneas de instrucciones IL versus cantidad de líneas de código para Photon, Quark y Spongent.

5.1.3 Memoria empleada

Con respecto a la complejidad espacial, se tomaron diferentes instantáneas del uso de la memoria con la herramienta *Diagnostic Tools* incluida en el ambiente de desarrollo Visual Studio 2019. Para ambos, utilizando la misma cadena de entrada "Hello World!", los resultados son similares. La memoria privada utilizada por el proceso, es decir *Heap, Stack, Memoria Virtual,* excluyendo memoria compartida con otros procesos, es aproximadamente 9 MegaBytes para ambos algoritmos. Similares resultados se obtienen al medir el tamaño del *Heap* en distintos puntos de interés: Para Photon, se tomaron instantáneas al pausar el proceso en las funciones: *Init, Padding, CompressFunction, Squeeze,* y previo a la visualización del digesto obtenido. Para Quark, se interrumpió en: *Init, Update, Final* y *PrintDigest.* El rango de valores para Quark se lo observó entre 64,80 y 66,81 KiloBytes entre la primer y última detención. Para Photon, el rango obtenido es de 64,63 y 65,28 KB. Para Spongent, se registraron valores al detener el proceso previo a la llamada de la función *spongent*, previo al llamado de la operación de permutación y al completar la función hash, registrando 67.90 KB previo al

inicio, 68,33 KB previo a cada llamada de permutación y 68,30 KB al terminar todo el procesamiento del digesto. Es así que Photon aventaja a Quark y a Spongent respecto al espacio consumido en memoria (ver Figuras 16, 17 y 18).

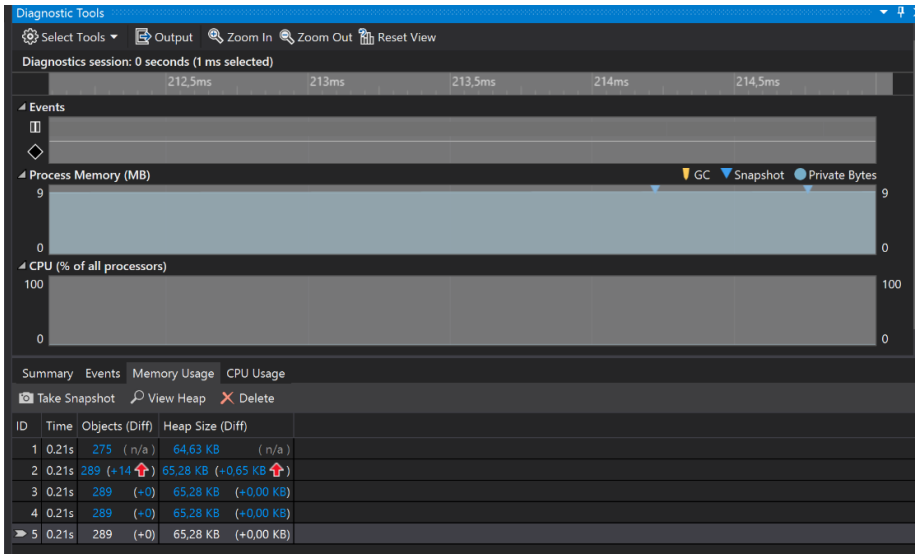


Figura 16. Instantáneas de la Herramienta de diagnóstico de Visual Studio 2019, incluyendo tamaño de la memoria Heap en cinco instantes distintos para Photon.

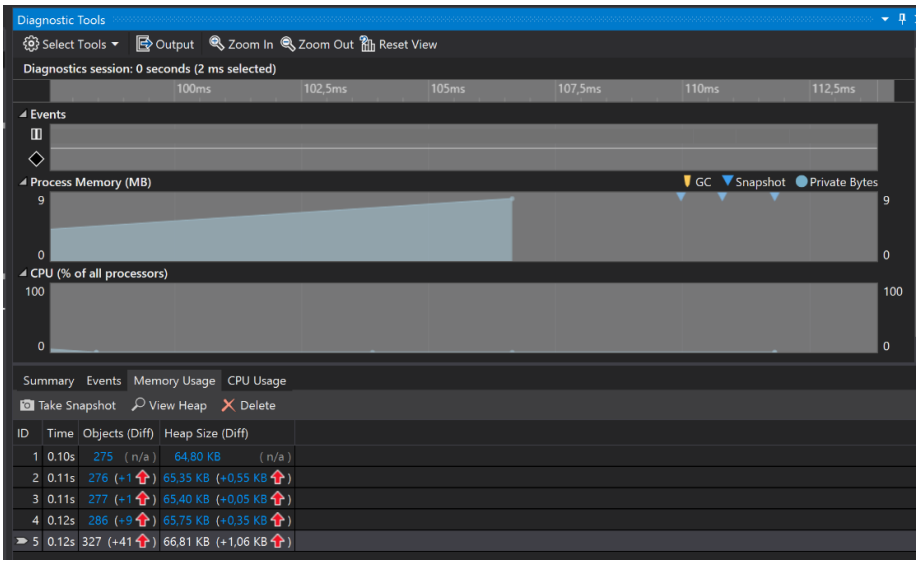


Figura 17. Instantáneas de la Herramienta de diagnóstico de Visual Studio 2019, incluyendo tamaño de la memoria Heap en cinco instantes distintos para Quark.

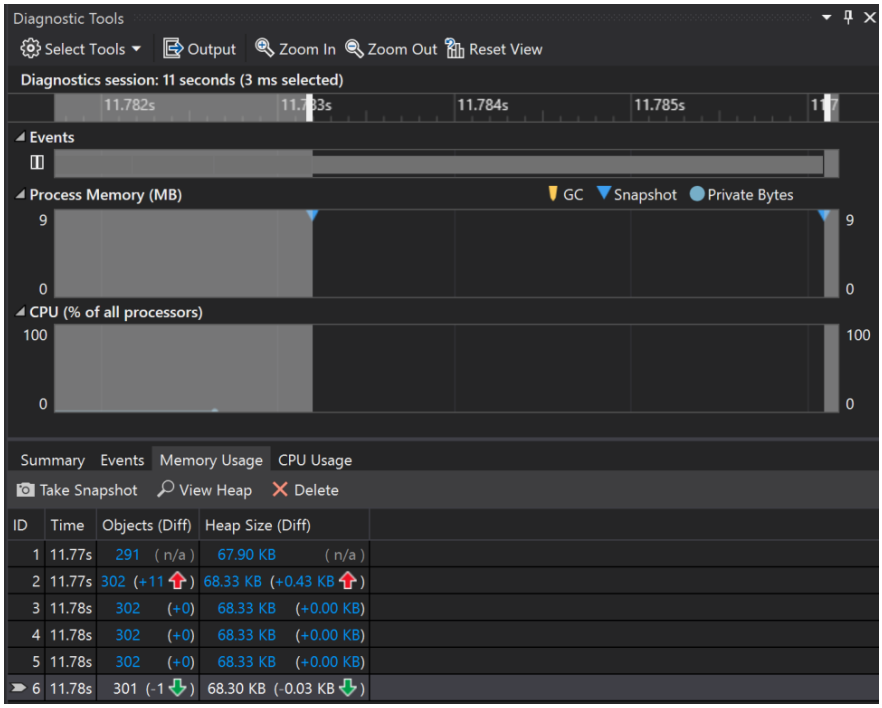


Figura 18. Instantáneas de la Herramienta de diagnóstico de Visual Studio 2019, incluyendo tamaño de la memoria Heap en seis instantes distintos para Spongent.

Como método adicional para evaluar la memoria privada empleada, se registró la misma mediante la clase del sistema *System.Diagnostics.Process* mediante la propiedad *PrivateMemorySize64* (Microsoft Docs. 2021), la cual registra la memoria empleada en *bytes*. Photon reportó utilizar 9.039.872 bytes en promedio, mientras que Quark registró 9.310.208 bytes. El primer lugar es para Spongent, utilizando 9.023.488 bytes. Estos valores son congruentes con el valor reportado por la herramienta de diagnóstico de Visual Studio en el apartado *Process Memory*, siendo Spongent el menos demandante en cuestión de memoria empleada.

De esta manera, se demuestra que Photon logra aventajar a Quark en mejores tiempos de ejecución, menor cantidad de instrucciones de bajo nivel y menor utilización de memoria empleada, superando ampliamente en tiempos de ejecución a Spongent a pesar de que este último registra menor cantidad de memoria consumida en la herramienta provista por Visual Studio, la cual no es ampliamente significativa. En líneas generales, se demuestra la hipótesis posicionando a Photon como la función hash más eficiente de las seleccionadas.

5.2 Framework de pruebas en dispositivo Raspberry Pi

Habiendo demostrado la hipótesis planteada en un entorno de pruebas de alto rendimiento, se procede a evaluar los algoritmos desarrollados en un ambiente de escasos recursos. El dispositivo elegido es el computador de bajo costo Raspberry Pi 3 Model B+, el cual cuenta con un procesador ARMv8 de 64 bits a 1.4 GHz y con 1GB de memora SDRAM LPDDR2 (ver Figura 19).

Figura 19. Dispositivo Raspberry Pi 3 Model B+ montado dentro de una carcasa (Fuente: Fotografía propia).

Se procede a instalar el sistema operativo compatible con el dispositivo. Se eligió un sistema operativo Linux acondicionado para este dispositivo: Ubuntu Server for ARM 20.04.3 LTS de 64 bits. Esta versión asegura tener un soporte a largo plazo, y contar con las características mínimas para consumir la menor cantidad de recursos posibles. Se instala en una tarjeta micro SD de 16 GB a través de la herramienta Raspberry Pi Imager.

Teniendo el sistema operativo en funcionamiento, se instala el *Software Development Kit* del *framework .NET Core 2.1* («Download .NET Core 2.1 SDK (v2.1.818) - Linux Arm64 Binaries» 2021), adecuado para el uso del procesador ARM64. Para ello, se crea un directorio en la ubicación */usr/share/dotnet,* y se descomprime el archivo descargado en dicha ubicación. Luego, se crea un enlace simbólico al directorio de ejecutables */usr/bin/dotnet* (ver Figura 20).

```
sudo mkdir -p /usr/share/dotnet
sudo tar dotnet21.tar.gz -C /usr/share/dotnet
sudo ln -s /usr/share/dotnet/dotnet /usr/bin/dotnet
```

Figura 20. Comandos para la instalación de .Net Core Framework 2.1 en Linux.

Finalmente, se comprueba la instalación con el comando *dotnet --version,* el cual despliega en pantalla el número 2.1.818 concordante con la versión descargada.

Se conecta el dispositivo a un enrutador, asignándosele una dirección IP. Se accede remotamente al dispositivo mediante *Secure Shell* o *SSH.* Para copiar los archivos compilados, se utiliza el comando *rsync* desde el dispositivo de origen (ver Figura 21).

```
ssh ubuntu@192.168.0.245
rsync -a hash-rasp-arm64 ubuntu@192.168.0.245:/home/ubuntu/
```

Figura 21. Comandos en Linux para la conexión y sincronización de directorios entre dos dispositivos.

Los mismos fueron recompilados, especificando manualmente el identificador correspondiente *Linux-arm64* en la etiqueta *RuntimeIdentifier* dentro del archivo *FolderProfile.pubxml.*

Se ejecutan los archivos de pruebas de rendimiento mediante el comando *nice* que establece la ejecución con la máxima prioridad, la cual se establece con un valor de -20 (ver Figura 22).

```
sudo nice –n -20 dotnet /home/Ubuntu/photoncs-profiler-rasp.dll
```

Figura 22. Comando en Linux para la ejecución de una aplicación .Net Core con máxima prioridad.

Los resultados obtenidos se especifican a continuación.

5.2.1 Tiempo de ejecución

Los resultados muestran tiempos de ejecución en el equipo Raspberry Pi que son 5,4 veces más lentos en promedio para el algoritmo Photon, 19 veces más lentos para el algoritmo Quark y 5 veces más lentos para Spongent que aquellos obtenidos en el equipo de alto rendimiento, manteniendo la linealidad entre caracteres de entrada y tiempo de ejecución. El primero alcanza el hash/s pasados los 10.000 caracteres de entrada, mientras que el segundo y el tercero los alcanza superando los 2.000 caracteres de entrada (ver Tablas V, VI y VII).

Tabla V. Registro de valores temporales y tasa de hashes para el algoritmo Photon en Raspberry Pi 3 Model B+.

No.	Longitud Cadena de Entrada [carác.]	Tiempo ejecucion 1 entrada [s]	Tiempo ejecucion 1 ciclo [s]	Tiempo promedio ejec. por entrada [s]	Tasa de Hashes / s
0	0	0,0011027	0,5875487	0,00117510	850,993
1	1	0,0010837	0,5500852	0,00110017	908,950
2	2	0,0010982	0,5481778	0,00109636	912,113
3	3	0,0013240	0,6588506	0,00131770	758,897
4	4	0,0013026	0,6587977	0,00131760	758,958
5	5	0,0015359	0,7658592	0,00153172	652,862
6	6	0,0015360	0,7667146	0,00153343	652,133
7	7	0,0015271	0,7678576	0,00153572	651,162
8	8	0,0017770	0,8769655	0,00175393	570,148
9	9	0,0017415	0,8763254	0,00175265	570,564
10	10	0,0019958	0,9869586	0,00197392	506,607
11	20	0,0028257	1,4232530	0,00284651	351,308
12	30	0,0037258	1,8600258	0,00372005	268,813
13	40	0,0045974	2,2973113	0,00459462	217,646
14	50	0,0058765	2,7325698	0,00546514	182,978
15	60	0,0063395	3,1704470	0,00634089	157,706
16	70	0,0071834	3,6059542	0,00721191	138,660
17	80	0,0080718	4,0446521	0,00808930	123,620
18	90	0,0089819	4,4840800	0,00896816	111,506
19	100	0,0098250	4,9193952	0,00983879	101,639
20	200	0,0185526	9,2969557	0,01859391	53,781
21	300	0,0272715	13,6633288	0,02732666	36,594
22	400	0,0359672	18,0271261	0,03605425	27,736
23	500	0,0446955	22,4013138	0,04480263	22,320
24	600	0,0534411	26,7723404	0,05354468	18,676
25	700	0,0621926	31,1381420	0,06227628	16,057
26	800	0,0709331	35,5155493	0,07103110	14,078
27	900	0,0795896	39,8962076	0,07979242	12,533
28	1.000	0,0883599	44,2577296	0,08851546	11,297
28	1.000	0,0884217	0,4425839	0,08851678	11,297
29	10.000	0,8754253	4,3808638	0,87617276	1,141
30	100.000	8,7479536	43,7410030	8,74820060	0,114
31	1.000.000	87,4812984	437,4289862	87,48579724	0,011

Tabla VI. Registro de valores temporales y tasa de hashes para el algoritmo Quark en Raspberry Pi 3 Model B+.

No.	Longitud Cadena de Entrada [carác.]	Tiempo ejecucion 1 entrada [s]	Tiempo ejecucion 1 ciclo [s]	Tiempo promedio ejec. por entrada [s]	Tasa de Hashes / s
0	0	0,0066078	0,7355400	0,00735540	135,955
1	1	0,0069551	0,7093664	0,00709366	140,971
2	2	0,0073602	0,7497762	0,00749776	133,373
3	3	0,0077156	0,7878544	0,00787854	126,927
4	4	0,0081303	0,8273783	0,00827378	120,864
5	5	0,0085420	0,8665055	0,00866506	115,406
6	6	0,0093364	0,9080287	0,00908029	110,129
7	7	0,0092543	0,9470672	0,00947067	105,589
8	8	0,0096522	0,9875244	0,00987524	101,263
9	9	0,0100047	1,0241202	0,01024120	97,645
10	10	0,0104452	1,0656816	0,01065682	93,837
11	20	0,0143368	1,4600223	0,01460022	68,492
12	30	0,0186106	1,8506370	0,01850637	54,035
13	40	0,0224004	2,2419110	0,02241911	44,605
14	50	0,0258693	2,6414706	0,02641471	37,858
15	60	0,0296955	3,0336000	0,03033600	32,964
16	70	0,0340302	3,4276794	0,03427679	29,174
17	80	0,0380214	3,8224594	0,03822459	26,161
18	90	0,0414768	4,2175089	0,04217509	23,711
19	100	0,0456023	4,6089571	0,04608957	21,697
20	200	0,0847759	8,5455040	0,08545504	11,702
21	300	0,1235764	12,4774089	0,12477409	8,014
22	400	0,1618843	16,4063662	0,16406366	6,095
23	500	0,2009407	20,3449074	0,20344907	4,915
24	600	0,2402199	24,2799942	0,24279994	4,119
25	700	0,2798256	28,2295858	0,28229586	3,542
26	800	0,3176360	32,1761887	0,32176189	3,108
27	900	0,3577765	36,1253709	0,36125371	2,768
28	1.000	0,3965503	40,0682256	0,40068226	2,496
29	10.000	3,9053497	23,4338182	4,68676364	0,213
30	100.000	39,0033180	234,0354588	46,80709176	0,021
31	1.000.000	389,5217238	2337,7475685	467,54951370	0,002

Tabla VII. Registro de valores temporales y tasa de hashes para el algoritmo Spongent en Raspberry Pi 3 Model B+.

No.	Longitud Cadena de Entrada [carác.]	Tiempo ejecucion 1 entrada [s]	Tiempo ejecucion 1 ciclo [s]	Tiempo promedio ejec. por entrada [s]	Tasa de Hashes / s
0	1	0.0059799	0.1257558	0.00628779	159.038390
1	2	0.0057935	0.1192134	0.00596067	167.766375
2	3	0.0056908	0.1202630	0.00601315	166.302188
3	4	0.0063490	0.1346710	0.00673355	148.510073
4	5	0.0063877	0.1339380	0.00669690	149.322821
5	6	0.0070735	0.1492719	0.00746360	133.983690
6	7	0.0070736	0.1487012	0.00743506	134.497906
7	8	0.0077849	0.1641176	0.00820588	121.863834
8	9	0.0077544	0.1639037	0.00819519	122.022871
9	10	0.0084869	0.1789174	0.00894587	111.783426
10	20	0.0120502	0.2535082	0.01267541	78.892912
11	30	0.0155876	0.3278188	0.01639094	61.009314
12	40	0.0193999	0.4022391	0.02011196	49.721671
13	50	0.0226080	0.4761592	0.02380796	42.002759
14	60	0.0265826	0.5507004	0.02753502	36.317388
15	70	0.0312877	0.6252513	0.03126257	31.987139
16	80	0.0332078	0.7003863	0.03501932	28.555670
17	90	0.0367062	0.7749292	0.03874646	25.808809
18	100	0.0402386	0.8489580	0.04244790	23.558291
19	200	0.0755217	1.5935804	0.07967902	12.550355
20	300	0.1114089	2.3392396	0.11696198	8.549787
21	400	0.1466782	3.0815671	0.15407836	6.490204
22	500	0.1820073	3.8271452	0.19135726	5.225827
23	600	0.2175849	4.5729580	0.22864790	4.373537
24	700	0.2529505	5.3160876	0.26580438	3.762165
25	800	0.2885693	6.0619794	0.30309897	3.299252
26	900	0.3234977	6.8062372	0.34031186	2.938481
27	1.000	0.3592274	7.5489443	0.37744722	2.649377
28	10.000	3.5509793	71.2550124	3.56145274	0.140341
29	100.000	35.4470846	708.8084613	70.87612249	0.014109
30	1.000.000	354.3899502	7089.4810539	708.8084114	0.001410

5.2.2 Tamaño de código

Los resultados son idénticos a los obtenidos en la sección previa, dado que la cantidad de instrucciones se obtienen a partir del código fuente, el cual es el mismo independientemente de la plataforma a la cual esté compilado el archivo.

El impacto en el almacenamiento es de 10,5 KB para el archivo *photoncs.dll*, de 9 KB para el archivo *quarkcs.dll* y de 8 KB para *spongentcs.dll*, aventajando éste último en este aspecto a sus contrapartes. Se confirma que Spongent logra reducir el impacto del uso del almacenamiento del dispositivo.

5.2.3 Memoria empleada

La memoria empleada reportada por el comando *ps aux* (Brown 2021) en plena ejecución es de 2.542.416 kilobytes para la memoria virtual, 38.292 kilobytes para el tamaño del conjunto residente RSS. La primera indica memoria que el proceso puede utilizar, ya sea que haya entrado en *swap*, memoria alojada y no utilizada, como así también memora compartida. Mientras que la RSS incluye *heap* y *stack* y memoria compartida. Por otra parte, no se logró obtener resultados mediante la clase de diagnóstico *System.Diagnostics.Process,* ya que la misma arrojó un valor de cero, probablemente por incompatibilidad con la plataforma. No se pudo determinar la memoria empleada con exactitud, por tratarse de valores de memoria que pueden ser compartidas y por ende, contarse más de una vez los valores obtenidos de memoria utilizada para varios procesos distintos.

6. Conclusiones

Debido a que existen diversas configuraciones para un mismo algoritmo, en el presente proyecto se tomaron las configuraciones más livianas de cada uno, haciendo foco en el rendimiento al procesar un mensaje de entrada como primer criterio de evaluación. Esta elección se hizo en carácter de favorecer las configuraciones menos demandantes para poder comparar cada algoritmo con su contraparte al mismo nivel de prestaciones. Los autores detallan que cada configuración tiene como objetivo un uso diferente, donde configuraciones más robustas pierden en velocidad de ejecución, pero ganan en seguridad dependiendo de lo que el desarrollador considere apropiado para priorizar en su dispositivo.

Los algoritmos livianos seleccionados fueron diseñados originalmente considerando su implementación en hardware y tratando de minimizar el impacto de uso del procesador utilizando la métrica GE o *Gate Equivalence* (Equivalencia de Compuertas) para medir dicho impacto. Es así como su implementación en software original utiliza código con punteros y administración de memoria. La métrica GE se desestima para el presente proyecto por quedar fuera del alcance, ya que es una métrica que mide la complejidad de un circuito electrónico. Si

bien C# permite el uso de código denominado “inseguro” mediante la palabra reservada *unsafe*, se descartó esa posibilidad para realizar una implementación en software sin manejo de memoria.

El lenguaje C# presentó limitaciones en las operaciones lógicas y de corrimiento de bits, ya que el mismo solo trabaja con operandos definidos como *int* o superior. Toda variable de longitud inferior a *int* será aplicada una conversión de tipo a *int.* Esto imposibilitó trabajar con variables definidas como *byte* en el algoritmo Spongent. Se logró saltear dicha dificultad realizando las operaciones de bits correspondientes, y posteriormente agregar una operación AND con el valor 0xFF para obtener el valor reducido y convertirlo al tamaño requerido *byte*.

Dado que en la implementación original de Quark, en la función Update se podría acceder a posiciones que sobrepasaran el tamaño de la longitud de la cadena de entrada (en cuyo caso se devuelve un 0), fue necesario agregar una condición mediante el operador ternario “?” para poder devolver el byte 0 cuando se requiera acceder a posiciones que superen la longitud del vector. Si bien C# inicializa el vector con el valor 0 para todas las posiciones por defecto, y es posible crear arreglos con tamaño dinámico, de esta manera se logra tener la misma implementación con un tamaño del vector igual a la cantidad de caracteres del mensaje de entrada y sin utilizar un vector dinámico.

Se realizaron pruebas de entrada con caracteres orientales codificados en Unicode UTF-8. Debido a que los caracteres pueden ser representados utilizando de 1 a 4 bytes por carácter, los algoritmos respondieron de distintas maneras: Photon detuvo su ejecución por desbordamiento. Spongent entregó el mismo digesto para cadenas distintas. Quark logró procesar la entrada sin problemas aparentes.

La cadena de entrada vacía fue procesada en Photon y Quark, no así en Spongent. Es conveniente que el algoritmo procese o tenga predefinido una salida para una entrada vacía, ya que hay casos donde el procesamiento de una entrada vacía amerita una salida. Por ejemplo, cuando se obtienen los digestos a partir de distintos archivos, es correcto marcar dos archivos vacíos con distinto nombre como archivos iguales, puesto que ambos están vacíos y deben estar relacionados con un mismo digesto.

Photon demostró ser el algoritmo más eficiente en tiempos de procesamiento y memoria empleada. Si bien Spongent logró minimizar el espacio utilizado en memoria física y cantidad

de instrucciones, su costo temporal por operación hace que no sea factible su utilización con esta implementación.

Con respecto a la seguridad, los autores de Photon (Guo 2020; Guo, Peyrin y Poschmann 2011) hacen referencia a que las longitudes de los digestos para dispositivos conectados pueden variar entre 64 y 80 bits, según lo requerido en los casos de uso de etiquetas RFID. Se trata de mantener un balance entre seguridad y performance, el cual se puede ajustar a través de las distintas versiones ofrecidas por los autores.

El Instituto Nacional de Estándares y Tecnología de Estados Unidos continúa desarrollando su proceso de estandarización de criptografía liviana (Sonmez Turan et al. 2021). Actualmente se encuentra en la segunda etapa, donde fueron seleccionados los últimos 10 finalistas. Se hace hincapié en la seguridad que puedan brindar los algoritmos, pero también se destaca que no se verán favorecidos aquellos candidatos que no logren superar los actuales estándares vigentes en cuanto a la performance que puedan brindar. Aquellos grupos que lograron compensar sus decisiones de diseño entre performance y seguridad de forma flexible, y que pudieran abarcar un amplio espectro de casos de uso, fueron considerados prioritariamente. También se tuvo en cuenta la diversidad de los algoritmos, agrupándolos por sus primitivas subyacentes y seleccionando los mejores candidatos de esos subgrupos similares. Con respecto a la seguridad post-cuántica, el Instituto no hizo ningún requerimiento formal al respecto para criptografía liviana (se encuentra llevando a cabo su propio proceso de estandarización aparte). Sin embargo, algunos candidatos presentaron versiones de sus algoritmos resistentes a ataques cuánticos. La tercera ronda de evaluación tendrá una duración de 12 meses y dará paso al proceso de estandarización de criptografía liviana.

7. Bibliografía

ASHTON, K., 2009. That «Internet of Things» Thing. *RFID journal 22.7 (2009): 97-114.*, pp. 1.

ATZORI, L., IERA, A. y MORABITO, G., 2016. Understanding the Internet of Things: definition, potentials, and societal role of a fast evolving paradigm. *Ad Hoc Networks*, vol. 56. DOI 10.1016/j.adhoc.2016.12.004.

AUMASSON, J.-P., 2020. *Quark Lightweight cryptographic hash functions (reference code)* [en línea]. C. S.l.: s.n. [Consulta: 1 noviembre 2021]. Disponible en: https://github.com/veorq/Quark.

AUMASSON, J.-P., HENZEN, L., MEIER, W. y NAYA-PLASENCIA, M., 2010. Quark: A Lightweight Hash. En: S. MANGARD y F.-X. STANDAERT (eds.), *Cryptographic Hardware and Embedded Systems, CHES 2010*. Berlin, Heidelberg: Springer, pp. 1-15. ISBN 978-3-642-15031-9. DOI 10.1007/978-3-642-15031-9_1.

BENVENUTO, C.J., 2012. Galois Field in Cryptography. , pp. 11.

BERTONI, G., DAEMEN, J., PEETERS, M. y VAN ASSCHE, G., 2011. Cryptographic sponge functions. [en línea]. [Consulta: 19 agosto 2021]. Disponible en: https://keccak.team/files/CSF-0.1.pdf.

BOGDANOV, A., KNEŽEVIĆ, M., LEANDER, G., TOZ, D., VARICI, K. y VERBAUWHEDE, I., 2011. *spongent: A Lightweight Hash Function*. S.l.: s.n. ISBN 978-3-642-23950-2.

BROWN, K., 2021. How to use ps command in Linux: Beginners guide. *Linux Tutorials - Learn Linux Configuration* [en línea]. [Consulta: 24 octubre 2021]. Disponible en: https://linuxconfig.org/how-to-use-ps-command-beginners-guide.

BUCHANAN, W.J., LI, S. y ASIF, R., 2017. Lightweight cryptography methods. *Journal of Cyber Security Technology*, vol. 1, no. 3-4, pp. 187-201. ISSN 2374-2917, 2374-2925. DOI 10.1080/23742917.2017.1384917.

CANNIÈRE, C., DUNKELMAN, O. y KNEŽEVIĆ, M., 2009. *Katan and Ktantan —A Family of Small and Efficient Hardware-Oriented Block Ciphers*. S.l.: s.n.

COMPUTER SECURITY DIVISION, I.T.L. y NIST, 2017. Lightweight Cryptography | CSRC | CSRC. *CSRC | NIST* [en línea]. [Consulta: 4 junio 2021]. Disponible en: https://csrc.nist.gov/projects/lightweight-cryptography.

COSKUN, B. y MEMON, N., 2006. Confusion/Diffusion Capabilities of Some Robust Hash Functions. *Information Sciences and Systems, 2006 40th Annual Conference on*. S.l.: s.n., pp. 1188-1193. DOI 10.1109/CISS.2006.286645.

DAEMEN, J. y RIJMEN, V., 1999. The Rijndael Block Cipher. , pp. 45.

DAVIS, R., 1978. The data encryption standard in perspective. *IEEE Communications Society Magazine*, vol. 16, no. 6, pp. 5-9. ISSN 0148-9615. DOI 10.1109/MCOM.1978.1089771.

DELFS, H. y KNEBL, H., 2015. *Introduction to Cryptography* [en línea]. Berlin, Heidelberg: Springer Berlin Heidelberg. [Consulta: 3 junio 2021]. Information Security and Cryptography. ISBN 978-3-662-47973-5. Disponible en: http://link.springer.com/10.1007/978-3-662-47974-2.

Download .NET Core 2.1 SDK (v2.1.818) - Linux Arm64 Binaries. *Microsoft* [en línea], 2021. [Consulta: 23 octubre 2021]. Disponible en: https://dotnet.microsoft.com/download/dotnet/thank-you/sdk-2.1.818-linux-arm64-binaries.

GROOT, W. de, 2019. *Spongent-avr Size-optimized Spongent hashing algorithm for ATtiny45* [en línea]. Assembly. S.l.: s.n. [Consulta: 1 noviembre 2021]. Disponible en: https://github.com/weedegee/spongent-avr.

GUO, J., 2020. Downloads - The PHOTON Family of Lightweight Hash Functions. [en línea]. [Consulta: 1 noviembre 2021]. Disponible en: https://sites.google.com/site/photonhashfunction/downloads.

GUO, J., PEYRIN, T. y POSCHMANN, A., 2011. The PHOTON Family of Lightweight Hash Functions. En: P. ROGAWAY (ed.), *Advances in Cryptology – CRYPTO 2011* [en línea]. Berlin, Heidelberg: Springer Berlin Heidelberg, Lecture Notes in Computer Science, pp. 222-239. [Consulta: 4 junio 2021]. ISBN 978-3-642-22791-2. Disponible en: http://link.springer.com/10.1007/978-3-642-22792-9_13.

HAMMAD, B.T., JAMIL, N., RUSLI, M.E. y ZABA, M.R., 2017. A survey of Lightweight Cryptographic Hash Function. , vol. 8, no. 7, pp. 9.

HELL, M., JOHANSSON, T. y MEIER, W., 2007. Grain: a stream cipher for constrained environments. *Int. J. Wirel. Mob. Comput.*, DOI 10.1504/IJWMC.2007.013798.

HENDERSON, T., 2013. Cryptography and Complexity. , pp. 17.

IBARRA-ESQUER, J., GONZÁLEZ-NAVARRO, F., FLORES-RIOS, B., BURTSEVA, L. y ASTORGA-VARGAS, M., 2017. Tracking the Evolution of the Internet of Things Concept Across Different Application Domains. *Sensors*, vol. 17, no. 6, pp. 1379. ISSN 1424-8220. DOI 10.3390/s17061379.

KENNEDY, J.B., 1926. When Woman is Boss. [en línea]. [Consulta: 5 junio 2021]. Disponible en: http://www.tfcbooks.com/tesla/1926-01-30.htm.

KERCKHOFFS, A., 1883. La cryptographic militaire. *Journal des Sciences Militaires*, vol. IX, pp. 5-38.

LINQPad - The .NET Programmer's Playground. [en línea], 2021. [Consulta: 15 octubre 2021]. Disponible en: https://www.linqpad.net/.

MARC STEVENS, ELIE BURSZTEIN, PIERRE KARPMAN, ANGE ALBERTINI y YARIK MARKOV, 2017. Announcing the first SHA1 collision. *Google Online Security Blog* [en línea]. [Consulta: 26 diciembre 2021]. Disponible en: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html.

MARWEDEL, P., 2010. Embedded and Cyber-Physical Systems in a Nutshell. ,

MICROSOFT DOCS., 2021. Process.PrivateMemorySize64 Property (System.Diagnostics). [en línea]. [Consulta: 23 octubre 2021]. Disponible en: https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.privatememorysize64.

MOUHA, N., 2015. The Design Space of Lightweight Cryptography. , pp. 20.

PAAR, C. y PELZL, J., 2009. *Understanding Cryptography: A Textbook for Students and Practitioners*. S.l.: Springer Science & Business Media. ISBN 978-3-642-04101-3.

SCHNEIER, B., 2013. Will Keccak = SHA-3? - Schneier on Security. [en línea]. [Consulta: 4 junio 2021]. Disponible en: https://www.schneier.com/blog/archives/2013/10/will_keccak_sha-3.html.

SHANNON, C.E., 1949. Communication theory of secrecy systems. [en línea]. [Consulta: 8 octubre 2021]. Disponible en: https://ieeexplore.ieee.org/document/6769090.

SONMEZ TURAN, M., MCKAY, K., CHANG, D., CALIK, C., BASSHAM, L., KANG, J. y KELSEY, J., 2021. Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process. [en línea]. S.l.: National Institute of Standards and Technology. [Consulta: 1 noviembre 2021]. Disponible en: https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8369.pdf.

SWENSON, 2018. NIST Issues First Call for 'Lightweight Cryptography' to Protect Small Electronics. En: Last Modified: 2018-04-18T13:12-04:00, *NIST* [en línea]. [Consulta: 4 junio 2021]. Disponible en: https://www.nist.gov/news-events/news/2018/04/nist-issues-first-call-lightweight-cryptography-protect-small-electronics.

VERBLE, J., 2014. The NSA and Edward Snowden: surveillance in the 21st century. *ACM SIGCAS Computers and Society*, vol. 44, no. 3, pp. 14-20. ISSN 0095-2737. DOI 10.1145/2684097.2684101.

WARREN, G., 2021a. Common Language Runtime (CLR) overview - .NET. [en línea]. [Consulta: 15 octubre 2021]. Disponible en: https://docs.microsoft.com/en-us/dotnet/standard/clr.

WARREN, G., 2021b. Managed Execution Process. [en línea]. [Consulta: 15 octubre 2021]. Disponible en: https://docs.microsoft.com/en-us/dotnet/standard/managed-execution-process.

WARREN, G., 2021c. What is managed code? [en línea]. [Consulta: 15 octubre 2021]. Disponible en: https://docs.microsoft.com/en-us/dotnet/standard/managed-code.

WEHBE, R., 2021a. *El algoritmo Sponge*. 2021. S.l.: s.n.

WEHBE, R., 2021b. *Securité des Systèmes d'Information - AES*. 2021. S.l.: s.n.

Anexo A: Códigos fuente

1. Photon

Constants.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace photoncs
{
    public static class Constants
    {
      public const int ROUND = 12;

      //_PHOTON80_
      public const int S = 4;
      public const int D = 5;
      public const int RATE = 20;
      public const int RATEP = 16;
      public const int DIGESTSIZE = 80;
      public static readonly byte[,] RC = new byte[Constants.D, 12] {
                { 1, 3, 7, 14, 13, 11, 6, 12, 9, 2, 5, 10 },
                { 0, 2, 6, 15, 12, 10, 7, 13, 8, 3, 4, 11 },
                { 2, 0, 4, 13, 14, 8, 5, 15, 10, 1, 6, 9 },
                { 7, 5, 1, 8, 11, 13, 0, 10, 15, 4, 3, 12 },
                { 5, 7, 3, 10, 9, 15, 2, 8, 13, 6, 1, 14 }

      };
      public const byte WORDFILTER = ((byte)1 << (byte)Constants.S) - (byte)1;

      public static readonly byte[] SBOX = new byte[16] { 12, 5, 6, 11, 9, 0, 10, 13,
3, 14, 15, 8, 4, 7, 1, 2 };

      public static readonly byte[,] MixColMatrix = new byte[Constants.D, Constants.D]
 {
                { 1, 2, 9, 9, 2 },
                { 2, 5, 3, 8, 13 },
                { 13, 11, 10, 12, 1 },
                { 1, 15, 2, 3, 14 },
                { 14, 14, 8, 5, 12 }
      };

      public const byte ReductionPoly = 0x3;
    }
}
```

Photon.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using static System.Math;

namespace photoncs
{
```

```
public partial class Program
{
  public static void hash(ref byte[] digest, string mess, int BitLen)
  {
    byte[,] state = new byte[Constants.D, Constants.D];
    byte[] padded = new byte[4];

    Init(ref state);
    int MessIndex = 0;
    byte[] str = Encoding.ASCII.GetBytes(mess);
    while (MessIndex <= (BitLen - Constants.RATE))
    {
      CompressFunction(state, str, MessIndex);
      MessIndex += Constants.RATE;
    }
    int i;
    double j;
    for (i = 0; i < (Math.Ceiling(Constants.RATE / 8.0) + 1); i++)
      padded[i] = 0;

    j = Ceiling( (Convert.ToDouble(BitLen - MessIndex)) / 8.0);
    for (i = 0; i < j; i++)
      padded[i] = Convert.ToByte(mess[(MessIndex / 8) + i]);
    padded[i] = 0x80;
    CompressFunction(state, padded, MessIndex & 0x7);
    Squeeze(state, digest);
  }

  public static void Init(ref byte[,] state)
  {
    byte[] presets = new byte[3];
    presets[0] = (Constants.DIGESTSIZE >> 2) & 0xFF;
    presets[1] = Constants.RATE & 0xFF;
    presets[2] = Constants.RATEP & 0xFF;

    WordXorByte(state, presets, 0, Constants.D * Constants.D - 24 / Constants.S, 24);
  }

 public static void WordXorByte(byte[,] state, in byte[] str, int BitOffSet, int WordO
ffSet, int NoOfBits)
  {
    int i = 0;
    while (i < NoOfBits)
    {
      int param1 = (WordOffSet + (i / Constants.S)) / Constants.D;
      int param2 = (WordOffSet + (i / Constants.S)) % Constants.D;
      byte tempByte1 = GetByte(str, BitOffSet + i, Min(Constants.S, NoOfBits - i));
      byte tempByte2 = (byte)(Constants.S - Min(Constants.S, NoOfBits - i));

      state[param1, param2] ^= (byte)(tempByte1 << tempByte2);
      i += Constants.S;
    }
  }

  public static byte GetByte(in byte[] str, int BitOffSet, int NoOfBits)
  {
    byte temp1 = str[BitOffSet >> 3];
    byte temp2 = (byte)(4 - (BitOffSet & 0x4));
```

```
  byte param1 = (byte)(temp1 >> temp2);
  return (byte)(param1 & Constants.WORDFILTER);
 }

 public static void CompressFunction(byte[,] state, in byte[] str, int BitOffSet)
 {
  WordXorByte(state, str, BitOffSet, 0, Constants.RATE);
  Permutation(state, Constants.ROUND);
 }

 public static void Permutation(byte[,] state, int R)
 {
  int i;
  for (i = 0; i < R; i++)
  {
   AddKey(state, i);
   SubCell(state);
   ShiftRow(state);
   MixColumn(state);
  }
 }

 public static void AddKey(byte[,] state, int round)
 {
  int i;
  for (i = 0; i < Constants.D; i++)
   state[i, 0] ^= Constants.RC[i, round];
 }

 public static void PrintState(byte[,] state)
 {
  int i, j;
  for (i = 0; i < Constants.D; i++)
  {
   for (j = 0; j < Constants.D; j++)
    Console.Write("{0:X2}", state[i, j]);

   Console.WriteLine();
  }
  Console.WriteLine();
 }

 public static void SubCell(byte[,] state)
 {
  int i, j;
  for (i = 0; i < Constants.D; i++)
   for (j = 0; j < Constants.D; j++)
    state[i, j] = Constants.SBOX[state[i, j]];
 }

 public static void ShiftRow(byte[,] state)
 {
  int i, j;
  byte[] tmp = new byte[Constants.D];

  for (i = 1; i < Constants.D; i++)
  {
   for (j = 0; j < Constants.D; j++)
```

```
      tmp[j] = state[i, j];
    for (j = 0; j < Constants.D; j++)
      state[i, j] = tmp[(j + i) % Constants.D];
  }
 }

 public static void MixColumn(byte[,] state)
 {
  int i, j, k;
  byte[] tmp = new byte[Constants.D];
  for (j = 0; j < Constants.D; j++)
  {
   for (i = 0; i < Constants.D; i++)
   {
    byte sum = 0;
    for (k = 0; k < Constants.D; k++)
     sum ^= FieldMult(Constants.MixColMatrix[i, k], state[k, j]);
    tmp[i] = sum;
   }
   for (i = 0; i < Constants.D; i++)
    state[i, j] = tmp[i];
  }
 }

 public static byte FieldMult(byte a, byte b)
 {
  byte x = a, ret = 0;
  int i;
  for (i = 0; i < Constants.S; i++)
  {
   if (Convert.ToBoolean((b >> i) & 1))
    ret ^= x;
   if (Convert.ToBoolean((x >> (Constants.S - 1)) & 1))
   {
    x <<= 1;
    x ^= Constants.ReductionPoly;
   }
   else
    x <<= 1;
  }
  return (byte)(ret & Constants.WORDFILTER);
 }

 public static void printDigest(byte[] digest)
 {
  int i;
  for (i = 0; i<Constants.DIGESTSIZE / 8; i++)
   Console.Write("{0:X}", digest[i]);
  Console.WriteLine();
 }

 public static void Squeeze(byte[,] state, byte[] digest)
 {
  int i = 0;
  while (true)
  {
   WordToByte(state, digest, i, Min(Constants.RATEP, Constants.DIGESTSIZE - i));
   i += Constants.RATEP;
```

```
      if (i >= Constants.DIGESTSIZE) break;
      Permutation(state, Constants.ROUND);
     }
    }


  public static void WordToByte(byte[,] state, byte[] str, int BitOffSet, int NoOfBits)
    {
     int i = 0;
     while (i < NoOfBits)
     {

  WriteByte(str, Convert.ToByte((state[i / (Constants.S * Constants.D) , (i / Constants
.S) % Constants.D] & Constants.WORDFILTER) >> (Constants.S - Min(Constants.S, NoOfBits
 - i))), BitOffSet + i, Min(Constants.S, NoOfBits - i));
      i += Constants.S;
     }
    }

    public static void WriteByte(byte[] str, byte value, int BitOffSet, int NoOfBits)
    {
     int ByteIndex = BitOffSet >> 3;
     int BitIndex = BitOffSet & 0x7;
     byte localFilter = Convert.ToByte((((byte)1) << NoOfBits) - 1);
     value &= localFilter;
     if (BitIndex + NoOfBits <= 8)
     {
      int temp = (byte)(~(localFilter << (8 - BitIndex - NoOfBits)));
      str[ByteIndex] &= (byte)temp;
      str[ByteIndex] |= Convert.ToByte(value << (8 - BitIndex - NoOfBits));
     }
     else
     {

  uint tmp = ((((uint)str[ByteIndex]) << 8) & 0xFF00) | (((uint)str[ByteIndex + 1]) & 0
xFF);
      tmp &= ~((((uint)localFilter) & 0xFF) << (16 - BitIndex - NoOfBits));
      tmp |= (((uint)(value)) & 0xFF) << (16 - BitIndex - NoOfBits);
      str[ByteIndex] = Convert.ToByte((tmp >> 8) & 0xFF);
      str[ByteIndex + 1] = Convert.ToByte(tmp & 0xFF);
     }
    }
  }
}
```

```
Program.cs

using System;
using System.Diagnostics;

namespace photoncs
{
 public partial class Program
 {
     static void Main(string[] args)
     {
      Console.WriteLine("Photon Lightweight Hash Function in C#. UADE - LMB.");
```

```
      TestVector1();
     }


     public static void TestVector1()
     {
      byte[] digest = new byte[Constants.DIGESTSIZE / 8];

      Process proc = Process.GetCurrentProcess();
      Console.WriteLine("Memory usage 1: {0}", proc.PrivateMemorySize64.ToString());

      Console.Write("Input your message: ");
      string mess = Console.ReadLine();
      Console.WriteLine();

      Console.WriteLine("Message: " + mess);
      Console.Write("Digest: ");
      hash(ref digest, mess, mess.Length * 8);
      printDigest(digest);

      Console.WriteLine("Memory usage 2: {0}", proc.PrivateMemorySize64.ToString());
      Console.ReadLine();
     }
 }
}
```

2. Quark

```
Constants.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace quarkcs
{
    public static class Constants
    {
        public const int MAXDIGEST = 48;

        /*UQUARK*/
        public const int RATE = 1;
        public const int WIDTH = 17;
        public static readonly byte[] IV = new byte[]
                  { 0xd8,0xda,0xca,0x44,0x41,0x4a,0x09,0x97,
                    0x19,0xc8,0x0a,0xa3,0xaf,0x06,0x56,0x44,0xdb
                  };
        public const int ROUNDS_U = 4 * 136;
        public const int N_LEN_U = 68;
        public const int L_LEN_U = 10;
        public const int DIGEST = WIDTH;
    }
}
```

Quark.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace quarkcs
{
 public partial class Program
 {
  public struct HashState
  {
   public int pos;
   public uint[] x;
  }

  public static void Quark(ref byte[] output, string input, ulong inlen)
  {
   HashState state = new HashState
   {
    x = new uint[Constants.WIDTH * 8]
   };
   Init(ref state);
   Update(ref state, input, inlen);
   Final(ref state, ref output);
  }

  public static void Init(ref HashState state)
  {
   int i;

   for (i = 0; i < 8 * Constants.WIDTH; ++i)
    state.x[i] = (uint)((Constants.IV[i / 8] >> (7 - (i % 8))) & 1);

   state.pos = 0;
  }

  public static void Update(ref HashState state, string input, ulong databytelen)
  {
   int i;

   byte[] bstr = Encoding.ASCII.GetBytes(input);
   int j = 0;

   while (databytelen > 0)
   {
    byte u = (j >= input.Length) ? (byte)0 : bstr[j];

    for (i = 8 * state.pos; i < 8 * state.pos + 8; ++i)
    {
     state.x[
        (8 * (Constants.WIDTH - Constants.RATE)) + i
         ] ^= Convert.ToUInt32((u >> (i % 8)) & 1);
    }

    j++;
    databytelen -= 1;
    state.pos += 1;
```

```
    if (state.pos == Constants.RATE)
    {
     Permute(ref state.x);
     state.pos = 0;
    }
   }
  }

  public static void Permute(ref uint[] x)
  {
   Permute_u(ref x);
  }

  private static void Permute_u(ref uint[] x)
  {
   uint[] xx, yy, lfsr;
   uint h;
   int i;

   xx   = new uint[(Constants.N_LEN_U + Constants.ROUNDS_U)];
   yy   = new uint[(Constants.N_LEN_U + Constants.ROUNDS_U)];
   lfsr = new uint[(Constants.L_LEN_U + Constants.ROUNDS_U)];

   for (i = 0; i < Constants.N_LEN_U; ++i)
   {
    xx[i] = x[i];
    yy[i] = x[i + Constants.N_LEN_U];
   }

   for (i = 0; i < Constants.L_LEN_U; ++i)
    lfsr[i] = 0xFFFFFFFF;

   for (i = 0; i < Constants.ROUNDS_U; ++i)
   {
    xx[Constants.N_LEN_U + i] = xx[i] ^ yy[i];
    xx[Constants.N_LEN_U + i] ^= xx[i + 9] ^ xx[i + 14] ^ xx[i + 21] ^ xx[i + 28] ^
     xx[i + 33] ^ xx[i + 37] ^ xx[i + 45] ^ xx[i + 52] ^ xx[i + 55] ^ xx[i + 50] ^

(xx[i + 59] & xx[i + 55]) ^ (xx[i + 37] & xx[i + 33]) ^ (xx[i + 15] & xx[i + 9]) ^

(xx[i + 55] & xx[i + 52] & xx[i + 45]) ^ (xx[i + 33] & xx[i + 28] & xx[i + 21]) ^
     (xx[i + 59] & xx[i + 45] & xx[i + 28] & xx[i + 9]) ^
     (xx[i + 55] & xx[i + 52] & xx[i + 37] & xx[i + 33]) ^
     (xx[i + 59] & xx[i + 55] & xx[i + 21] & xx[i + 15]) ^
     (xx[i + 59] & xx[i + 55] & xx[i + 52] & xx[i + 45] & xx[i + 37]) ^
     (xx[i + 33] & xx[i + 28] & xx[i + 21] & xx[i + 15] & xx[i + 9]) ^
     (xx[i + 52] & xx[i + 45] & xx[i + 37] & xx[i + 33] & xx[i + 28] & xx[i + 21]);

    yy[Constants.N_LEN_U + i] = yy[i];
    yy[Constants.N_LEN_U + i] ^= yy[i + 7] ^ yy[i + 16] ^ yy[i + 20] ^ yy[i + 30] ^
     yy[i + 35] ^ yy[i + 37] ^ yy[i + 42] ^ yy[i + 51] ^ yy[i + 54] ^ yy[i + 49] ^

(yy[i + 58] & yy[i + 54]) ^ (yy[i + 37] & yy[i + 35]) ^ (yy[i + 15] & yy[i + 7]) ^

(yy[i + 54] & yy[i + 51] & yy[i + 42]) ^ (yy[i + 35] & yy[i + 30] & yy[i + 20]) ^
     (yy[i + 58] & yy[i + 42] & yy[i + 30] & yy[i + 7]) ^
     (yy[i + 54] & yy[i + 51] & yy[i + 37] & yy[i + 35]) ^
```

```
      (yy[i + 58] & yy[i + 54] & yy[i + 20] & yy[i + 15]) ^
      (yy[i + 58] & yy[i + 54] & yy[i + 51] & yy[i + 42] & yy[i + 37]) ^
      (yy[i + 35] & yy[i + 30] & yy[i + 20] & yy[i + 15] & yy[i + 7]) ^
      (yy[i + 51] & yy[i + 42] & yy[i + 37] & yy[i + 35] & yy[i + 30] & yy[i + 20]);

     lfsr[Constants.L_LEN_U + i] = lfsr[i];
     lfsr[Constants.L_LEN_U + i] ^= lfsr[i + 3];


 h = xx[i + 25] ^ yy[i + 59] ^ (yy[i + 3] & xx[i + 55]) ^ (xx[i + 46] & xx[i + 55]) ^
(xx[i + 55] & yy[i + 59]) ^

 (yy[i + 3] & xx[i + 25] & xx[i + 46]) ^ (yy[i + 3] & xx[i + 46] & xx[i + 55]) ^ (yy[i
 + 3] & xx[i + 46] & yy[i + 59]) ^
      (xx[i + 25] & xx[i + 46] & yy[i + 59] & lfsr[i]) ^ (xx[i + 25] & lfsr[i]);

 h ^= xx[i + 1] ^ yy[i + 2] ^ xx[i + 4] ^ yy[i + 10] ^ xx[i + 31] ^ yy[i + 43] ^ xx[i
+ 56] ^ lfsr[i];

     xx[Constants.N_LEN_U + i] ^= h;
     yy[Constants.N_LEN_U + i] ^= h;
    }

    for (i = 0; i < Constants.N_LEN_U; ++i)
    {
     x[i] = xx[Constants.ROUNDS_U + i];
     x[i + Constants.N_LEN_U] = yy[Constants.ROUNDS_U + i];
    }
   }

   private static void Final(ref HashState state, ref byte[] output)
   {
    int i;
    int outbytes = 0;
    byte u;

    state.x[8 * (Constants.WIDTH - Constants.RATE) + state.pos * 8] ^= 1;

    Permute(ref state.x);

    for (i = 0; i < Constants.DIGEST; ++i)
     output[i] = 0;

    while (outbytes < Constants.DIGEST)
    {
     for (i = 0; i < 8; ++i)
     {

 u = Convert.ToByte(state.x[8 * (Constants.WIDTH - Constants.RATE) + i + 8 * (outbytes
 % Constants.RATE)] & 1);
      output[outbytes] ^= (byte)(u << (7 - i));
     }

     outbytes += 1;
     if (outbytes == Constants.DIGEST)
      break;

     if ((outbytes % Constants.RATE) == 0)
```

```
      {
       Permute(ref state.x);
      }
     }
    }
   }
}
```

Program.cs

```
using System;
using System.Diagnostics;

namespace quarkcs
{
    public partial class Program
    {
     static void Main(string[] args)
     {
      Console.WriteLine("Quark Lightweight Hash Function in C#. UADE - LMB.");
      TestVector1();
     }

     public static void TestVector1()
     {
      byte[] digest = new byte[Constants.MAXDIGEST];

      Process proc = Process.GetCurrentProcess();
      Console.WriteLine("Memory usage 1: {0}", proc.PrivateMemorySize64.ToString());

      Console.Write("Input your message: ");
      string mess = Console.ReadLine();
      Console.WriteLine("Message: " + mess);

      Console.Write("Digest: ");
      Quark(ref digest, mess, (ulong)mess.Length);
      printDigest(ref digest);

      Console.WriteLine("Memory usage 2: {0}", proc.PrivateMemorySize64.ToString());
      Console.ReadLine();
     }

     public static void printDigest(ref byte[] digest)
     {
      int i;
      for (i = 0; i < Constants.MAXDIGEST / 8; i++)
          Console.Write("{0:X}", digest[i]);
      Console.WriteLine();
     }
    }
}
```

3. Spongent

Constants.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace spongentcs
{
    public static class Constants
    {
        public static readonly byte[] S = new byte[16] { 0xe, 0xd, 0xb, 0x0, 0x2, 0x1,
 0x4, 0xf, 0x7, 0xa, 0x8, 0x5, 0x9, 0xc, 0x3, 0x6 };
        public static readonly ushort b = 88;
        public static readonly ushort B = 88 / 8;
        public static readonly ushort n = 88;
        public static readonly byte[] state = new byte[34];
    }
}
```

Spongent.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;

namespace spongentcs
{
    public partial class Program
    {
        public static byte nextValueForLfsr(byte lfsr1)
        {
            uint lfsr = Convert.ToUInt16(lfsr1);
            uint value = unchecked((lfsr << 1)) | (
                                    (
                                        unchecked((lfsr >> 1)) ^ unchecked((lfsr >> 2)
) ^ unchecked((lfsr >> 3)) ^ unchecked((lfsr >> 7))
                                    ) & 1
                                );
            return Convert.ToByte(value & 0xFF);
        }

        public static byte reverse(byte b)
        {
            b = Convert.ToByte((b & 0xF0) >> 4 | (b & 0x0F) << 4);
            b = Convert.ToByte((b & 0xCC) >> 2 | (b & 0x33) << 2);
            return Convert.ToByte((b & 0xAA) >> 1 | (b & 0x55) << 1);
        }

        public static void pLayer()
        {
            byte[] tmp = new byte[11];

            for (uint idx = 0; idx < (Constants.b - 1); ++idx)
            {
```

```
                byte bit = Convert.ToByte((Constants.state[idx / 8] >> (byte)(idx % 8)
) & 0x1);

                if (bit != 0)
                {
                    uint dest = Convert.ToUInt16(((long)idx * Constants.b / 4) % (Cons
tants.b - 1));
                    tmp[dest / 8] |= Convert.ToByte(1 << (int)dest % 8);
                }
            }

            tmp[Constants.B - 1] |= Convert.ToByte(Constants.state[Constants.B - 1] &
0x80);

            for (int i = 0; i < Constants.B; i++)
                Constants.state[i] = tmp[i];
        }

        public static void permute()
        {
            byte lfsr = 0x9e;
            do
            {
                Constants.state[0] ^= lfsr;
                Constants.state[Constants.B - 1] ^= reverse(lfsr);

                lfsr = nextValueForLfsr(lfsr);

                for (uint idx = 0; idx < Constants.B; ++idx)
                    Constants.state[idx] = Convert.ToByte(Constants.S[Constants.state[
idx] >> 4] << 4 | Constants.S[Constants.state[idx] & 0xF]);

                pLayer();
            } while (lfsr != 0xFF);
        }

        public static void spongent(string mess, byte[] output)
        {
            uint idx = 0;
            byte[] input = Encoding.ASCII.GetBytes(mess);

            if (input[idx] != 0)
            {
                while ((idx+1) >= input.Length ? false : true))
                {
                    Constants.state[0] ^= input[idx];
                    Constants.state[1] ^= input[idx + 1];

                    permute();
                    idx += 2;
                }
            }

            if (idx > input.Length ? false : true)
                Constants.state[0] ^= 0x80;
            else
            {
                Constants.state[0] ^= input[idx];
```

```
                Constants.state[1] ^= 0x80;
            }
            permute();

            for (uint idx2 = 0; idx2 < Constants.n / 8; idx2 += 2)
            {
                output[idx2] = Constants.state[0];
                output[idx2 + 1] = Constants.state[1];

                permute();
            }
        }
    }
}
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;

namespace spongentcs
{
    public partial class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Spongent Lightweight Hash Function in C#. UADE - LMB.")
;
            TestVector1();
        }

        public static void TestVector1()
        {
            Console.Write("Input your message: ");
            string mess = Console.ReadLine();

            byte[] output = new byte[32];

            Process currentProc = Process.GetCurrentProcess();
            long memoryUsed = currentProc.PrivateMemorySize64;

            Console.WriteLine("Memory usage 1: {0}", memoryUsed);
            spongent(mess, output);

            memoryUsed = currentProc.PrivateMemorySize64;
            Console.WriteLine("Memory usage 2: {0}", memoryUsed);

            Console.Write("Digest: ");
            PrintDigest(ref output);
        }

        public static void PrintDigest(ref byte[] output)
        {
            int i;
            for (i = 0; i < Constants.B + 1; i++)
```

```
                Console.Write("{0:X}", output[i]);
            Console.WriteLine();

        }
    }
}
```

4. Profiler

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

namespace profiler
{
    public partial class Program
    {
        public const int POS = 32;
        public const int CYCLES = 1;
        public static int[] msgLength = new int[POS];
        private static readonly Random random = new Random();

        static void Main(string[] args)
        {
            Console.WriteLine("Profiler - Lightweight Hash Function in C#. UADE - LMB.
");
            TestVector1();
        }

        public static string GetARandomString(int length)
        {
            const string chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
            return length == 0 ? "" : new string(Enumerable.Repeat(chars, length)
              .Select(s => s[random.Next(s.Length)]).ToArray());
        }

        public static string[] GetAllRandomStrings()
        {
            string[] rndStr = new string[POS];

            int i = 0;
            for (int j = 0; j < 10; j++)
            {
                msgLength[i] = j;
                rndStr[i] = GetARandomString(j);
                ++i;
            }

            for (int j = 10; j < 101; j = j + 10)
            {
                msgLength[i] = j;
                rndStr[i] = GetARandomString(j);
                ++i;
            }
```

```
            for (int j = 200; j < 1001; j += 100)
            {
                msgLength[i] = j;
                rndStr[i] = GetARandomString(j);
                ++i;
            }

            for (int j = 10000; j < 1000001; j = j * 10)
            {
                msgLength[i] = j;
                rndStr[i] = GetARandomString(j);
                ++i;
            }
            return rndStr;
        }
        public static void TestVector1()
        {
            string[] rndStrs = GetAllRandomStrings();
            double[] totalTime = new double[POS];
            double[] oneTime = new double[POS];
            double[] avgTime = new double[POS];
            double[] hashRate = new double[POS];

            byte[] photonDigest = new byte[photoncs.Constants.DIGESTSIZE / 8];
            byte[] quarkDigest = new byte[quarkcs.Constants.MAXDIGEST];
            byte[] spongentDigest = new byte[32];

            for (int i = 0; i < POS; ++i)
            {
                Console.WriteLine("Computing Hash Function {0} times for position
{1}: ", CYCLES, i);

                var startTime = DateTime.UtcNow;
                for (int j = 0; j < CYCLES + 1; ++j)
                {
          photoncs.Program.hash(ref photonDigest, rndStrs[i], rndStrs[i].Length * 8);
        //quarkcs.Program.Quark(ref quarkDigest, rndStrs[i], (ulong)rndStrs[i].Length);
        //spongentcs.Program.spongent(rndStrs[i], spongentDigest);
                }
                var endTime = DateTime.UtcNow;
                TimeSpan interval = endTime - startTime;
                totalTime[i] = interval.TotalSeconds;
                avgTime[i] = (interval.TotalSeconds / CYCLES);
                hashRate[i] = (CYCLES / interval.TotalSeconds);

                startTime = DateTime.UtcNow;

          photoncs.Program.hash(ref photonDigest, rndStrs[i], rndStrs[i].Length * 8);
        //quarkcs.Program.Quark(ref quarkDigest, rndStrs[i], (ulong)rndStrs[i].Length);
        //spongentcs.Program.spongent(rndStrs[i], output);

                endTime = DateTime.UtcNow;
                interval = endTime - startTime;
                oneTime[i] = interval.TotalSeconds;

            }
```

```
            Console.WriteLine("N.; MSGLGTH; ONETIME; TTLTME; AVGTIME; HASHRATE");
            for (int i = 0; i < POS; i++)
                Console.WriteLine("{0}; {1}; {2}; {3}; {4}; {5};", i, msgLength[i], on
eTime[i], totalTime[i], avgTime[i], hashRate[i]);

            Console.ReadLine();
        }
    }
}
```

Anexo B: Glosario

AEAD: Cifrado autenticado con datos asociados. Provee confidencialidad y autenticidad de los datos.

AES: Estándar de cifrado avanzado, también conocido como algoritmo Rijndael.

AND: Operación de "Y" o "conjunción lógica" entre dos cadenas binarias. Si ambos bits de cada posición de la cadena son 1, se devuelve un 1. De lo contrario, se devuelve un 0.

Arduino: Placa con sensores programable de código abierto.

Bit: Unidad básica de información representada con un 0 y un 1.

Byte: Unidad de información que consiste en 8 bits.

Digesto: Secuencia de dígitos resultado de aplicar una función hash.

Distancia de Hamming: Cantidad de posiciones de bit en la cual dos cadenas binarias difieren.

Gate Equivalent (GE): Unidad de medida de complejidad de un circuito eléctrico.

Hash: Comúnmente llamado al digesto de una función hash.

Heap: Memoria habilitada a un programa o proceso de la cual puede obtener pedazos de la misma para utilizarla en sus tareas.

IoT: Internet of Things o "Internet de las Cosas". Denominación a dispositivos de uso cotidiano conectados a internet.

JIT: "Just in Time compilation". Compilación de un programa durante su ejecución.

LFSR: "Linear Feedback Shift Register". Registro cuyos bits se mueven hacia la izquierda o derecha, y que toma una función lineal de un estado previo como entrada. La función generalmente es XOR.

Main: Método o función de inicio de un programa.

Mainframe: Computadora de uso industrial y corporativo utilizada para aplicaciones críticas a gran escala.

MD5: Función hash de 128 bits de digesto ampliamente utilizada, pero cuya seguridad ha sido comprometida.

Microcontrolador: Pequeña computadora ensamblada en un chip de circuito integrado. Cuenta con su propio procesador y memoria, destinado a sistemas embebidos.

Nibble: Unidad de información que consiste en 4 bits.

NIST: Instituto Nacional de Estándares y Tecnología de Estados Unidos de América.

NSA: Agencia de Seguridad Nacional de los Estados Unidos de América.

Padding: Relleno que se le aplica a un bloque en el proceso de obtención del digesto.

PDF: Formato de Documento Portable. Archivo de texto e imágenes con formato listo para ser impreso.

Pixel: Mínima unidad controlable de una imagen o pantalla.

RAM: Memoria volátil de acceso aleatorio.

Raspberry Pi: Mini computadora de bajo costo y tamaño reducido.

RFID: Etiqueta de Identificación de Radio Frecuencia que utiliza campos magnéticos para alimentarse y comunicarse. Su uso clásico es adherirla a la mercancía para su fácil identificación.

SHA-1: Algoritmo de Hash Seguro versión 1. Función hash de 160 bits creada por la NSA, publicado en 1995.

SHA-2: Algoritmo de Hash Seguro versión 2. Función hash de digestos de 224, 256, 384 y 512 bits de longitud. Publicado por la NSA en 2001.

SHA-3: Algoritmo de Hash Seguro versión 3, también conocido por el nombre Keccak, con tamaño de digesto arbitrario. Publicado por el NIST en 2016.

Sistemas Ciberfísicos: Sistema o computadora inteligente conectada a la red que interactúa con su ambiente mediante el uso de algoritmos.

Sistemas Embebidos: Computadora completa dedicada a una función concreta instalada dentro de un sistema mecánico o eléctrico.

Stack: Memoria accedida en forma de pila por un programa para almacenar estado local.

XOR: Operación lógica de “O exclusivo” entre dos cadenas binarias. Si ambos bits de cada posición son iguales, el resultado es 0. De lo contrario, es 1.