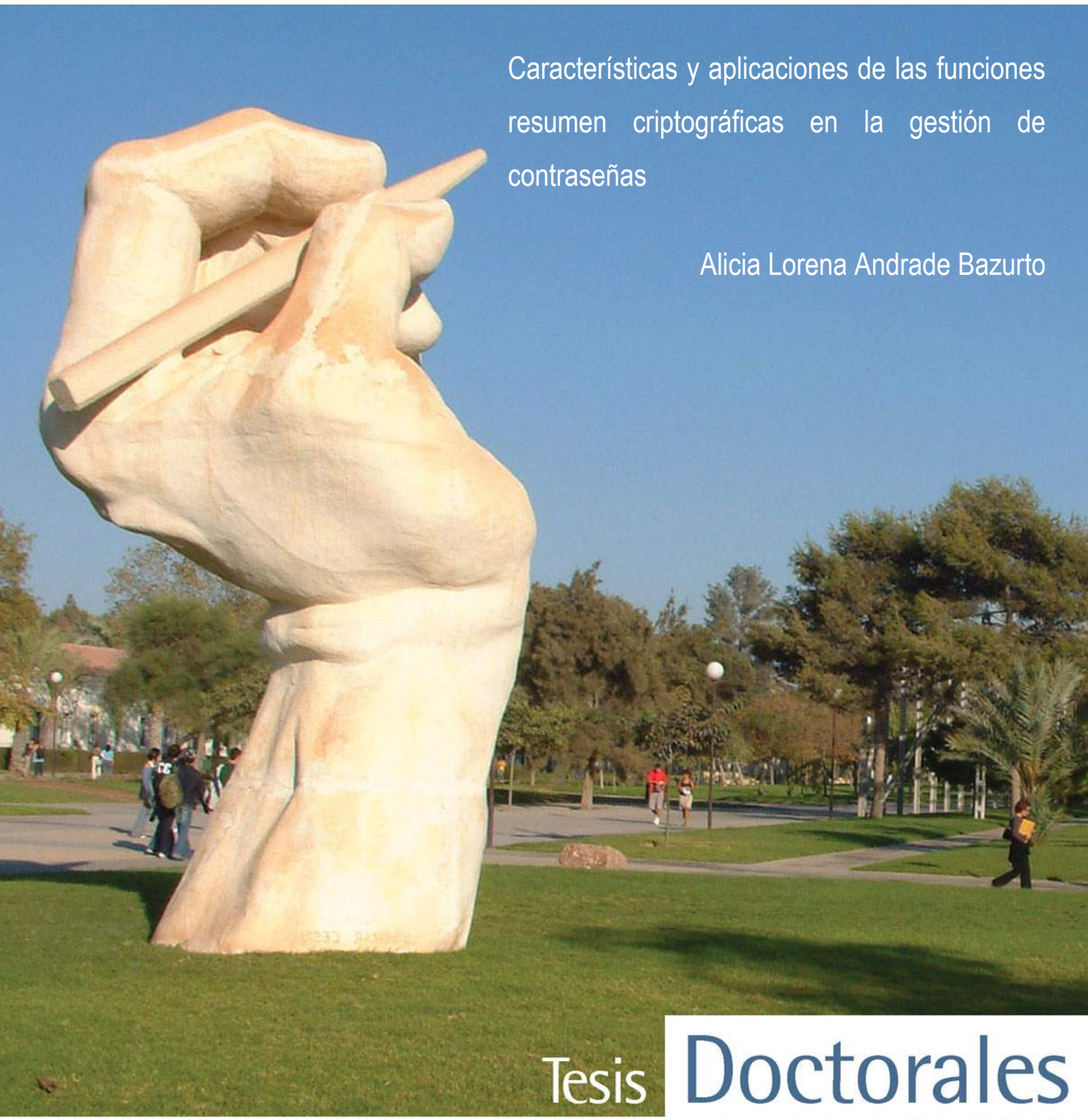
Universitat d'Alacant
Universidad de Alicante

Características y aplicaciones de las funciones resumen criptográficas en la gestión de contraseñas

Alicia Lorena Andrade Bazurto



Tesis Doctorales

www.eltallerdigital.com

UNIVERSIDAD de ALICANTE

Universitat d'Alacant
Universidad de Alicante

Instituto Universitario de Investigación en Informática
Escuela Politécnica Superior

Características y aplicaciones de las funciones resumen criptográficas en la gestión de contraseñas

ALICIA LORENA ANDRADE BAZURTO

Tesis presentada para aspirar al grado de
DOCTORA POR LA UNIVERSIDAD DE ALICANTE

DOCTORADO EN INFORMÁTICA

Dirigida por:
Dr. Rafael I. Álvarez Sánchez

Alicante, julio 2019

Índice

Índice de tablas

Índice de figuras

"Cada lugar donde nos sentimos seguros es un tesoro."

-Jan Jansen.

Agradecimiento

- *Especial gratitud al doctor Rafael Álvarez por su enseñanza y apoyo en la investigación; me llevo su amistad.*
- *A la Universidad Central del Ecuador y a la Universidad de Alicante con sus señores maestros, por hacerlo viable.*
- *A mi hijo Axel, mi regalo divino; por darme día a día la luz de su sonrisa.*
- *A mi esposo Franklin por el amor, apoyo y dedicación, y aún más durante mi ausencia en este trayecto.*
- *A mis padres (+) en su memoria todo mi esfuerzo, y a mis hermanos que me encaminaron en la vida.*

Resumen

Actualmente, la criptografía resulta de vital importancia en la protección de la información, garantizando la confidencialidad, autenticidad, integridad y disponibilidad. Dentro de esta área, las funciones resumen o hash criptográficas tienen mucha aplicabilidad en sistemas y protocolos seguros. Su función principal consiste en pasar de una cadena de longitud arbitraria (mensaje) a una de longitud fija (resumen) de forma que sea muy improbable obtener el mensaje a partir del resumen o encontrar dos mensajes que generen el mismo resumen.

Las funciones de derivación de claves basadas en contraseña (PBKDF), son funciones hash especializadas que se usan, comúnmente, para transformar las contraseñas de los usuarios en claves para el cifrado simétrico, así como para la autenticación de usuarios.

Se propone una PBKDF con tres niveles de optimización cuyo diseño se basa en emplear el estándar de cifrado avanzado (AES) como un generador pseudoaleatorio y aprovechar el soporte para la aceleración de hardware para AES para mitigar los ataques comunes a los sistemas de autenticación de usuarios basados en contraseña. Se analizan, también, sus características de seguridad, estableciendo que resulta equivalente a la seguridad de AES, y se compara su rendimiento con algoritmos PBKDF de prestigio, como Scrypt y Argon2, con resultados favorables.

Resum

Actualment, la criptografia resulta de vital importància en la protecció de la informació, garantint la confidencialitat, autenticitat, integritat i disponibilitat. Dins d'aquesta àrea, les funcions resum o hash criptogràfiques tenen molta aplicabilitat en sistemes i protocols segurs. La seua funció principal consisteix a passar d'una cadena de longitud arbitrària (missatge) a una de longitud fixa (resum) de manera que siga molt improbable obtenir el missatge a partir del resum o trobar dos missatges que generen el mateix resum.

Les funcions de derivació de claus basades en contrasenya (PBKDF), són funcions hash especialitzades que s'usen, comunament, per a transformar les contrasenyes dels usuaris en claus per al xifrat simètric, així com per a l'autenticació d'usuaris.

Es proposa una PBKDF amb tres nivells d'optimització el disseny de la qual es basa a emprar l'estàndard de xifrat avançat (AES) com un generador pseudoaleatorio i aprofitar el suport per a l'acceleració de hardware per a AES per a mitigar els atacs comuns als sistemes d'autenticació d'usuaris basats en contrasenya. S'analitzen, també, les seues característiques de seguretat, establint que resulta equivalent a la seguretat d'AES, i es compara el seu rendiment amb algorismes PBKDF de prestigi, com Scrypt i Argon2, amb resultats favorables.

Abstract

Nowadays, cryptography is of vital importance in the protection of information, guaranteeing confidentiality, authenticity, integrity and availability. Within this subject, cryptographic hash functions have many applications in secure systems and protocols. Their main function is to map from an arbitrary length string (message) to a fixed length string (hash or digest) so that it is very unlikely to obtain the message that corresponds to a given digest or to find two different messages that generate the same digest.

Password-based key derivation functions (PBKDF) are specialized hash functions commonly used to transform user passwords into keys for symmetric encryption as well as for user authentication.

We propose a PBKDF with three levels of optimization. Its design is based on using the Advanced Encryption Standard (AES) as a pseudo-random generator and leveraging hardware acceleration support for AES to mitigate common attacks to password-based user authentication systems. We also analyze its security features, establishing that it is equivalent to AES security, and its performance is compared with prestigious PBKDF algorithms, such as Scrypt and Argon2, achieving favorable results.

1 Introducción

La revolución tecnológica y el auge imparable de Internet han motivado que la sociedad se haya transformado en la llamada sociedad de la información, las empresas ofrecen sistemas de comercio electrónico, los gobiernos proporcionan servicios capaces de agilizar la interacción con la administración e incluso muchos usuarios particulares crean páginas web con temática diversa. Actualmente, se cumple el trigésimo aniversario de la creación del protocolo de transferencia de hipertexto (HTTP) por Tim Berners Lee (véase [1]) lo que, unido al uso generalizado y masivo de las tecnologías de la información, ha dado lugar al almacenamiento y transferencia de grandes flujos de información confidencial, cuya seguridad es transcendental.

A pesar de la funcionalidad ofrecida, estos sistemas pueden ser vulnerables a ataques maliciosos suponiendo, además de pérdidas económicas y de datos, un impacto significativo en la reputación de las empresas. Por ello, se requieren servicios con más seguridad, robustez y fiabilidad que se fundamentan en el uso de técnicas criptográficas y de seguridad de la información. Ambas son clave para lograr una comunicación segura entre pares, garantizando la privacidad, autenticidad, integridad y garantía de no repudio de la información.

Desde el inicio de las sociedades surgen las primeras técnicas orientadas al envío de información de manera privada, tanto criptográficas como de simple ocultamiento, ya que siempre ha existido la motivación de ocultar información a terceros, principalmente por la actividad militar y la necesidad de comunicarse de forma secreta (véase [2]).

Las personas pasan en línea buena parte de su tiempo, tanto por motivos profesionales como personales, desconociendo, en muchos casos, la existencia de ataques y amenazas que pueden perjudicar de forma directa a los usuarios en aspectos económicos, personales u otros (véase [3]). Consecuentemente, se debe mencionar que los ataques informáticos existen desde que existen los ordenadores; con software malicioso como el *ransomware* (véase [4]) que penetra y cifra archivos del sistema operativo, deshabilitando los equipos informáticos y solicitando un pago por su restauración.

Uno de los aspectos esenciales en la seguridad de los sistemas informáticos es la autentificación de la identidad de los usuarios mediante contraseñas, bien como mecanismo exclusivo o en combinación con otros factores como la biometría (véase [5]).

Tradicionalmente, las contraseñas se han almacenado en claro en una base de datos o archivo. Esto presentaba muchos problemas ya que, por lo general, los atacantes podían obtener dicha información con relativa facilidad.

Para mejorar la seguridad de los sistemas de autentificación por contraseña, el primer paso fue utilizar una función resumen criptográfica para procesar las contraseñas, de modo que no se puedan reconocer a simple vista en caso de que se comprometiera la base de datos con las contraseñas. Lamentablemente, los atacantes aún podían descubrir mucha información ya que la misma contraseña, aunque fuera empleada por usuarios distintos, siempre producía el mismo valor en la base de datos y era muy fácil compilar un diccionario de contraseñas frecuentes y acelerar aún más el proceso.

La siguiente mejora consistió en concatenar una cadena binaria aleatoria (llamada comúnmente *sal*) a la contraseña antes de calcular su resumen, evitando que la misma contraseña para usuarios distintos resultara en el mismo resumen en la base de datos e impidiendo ataques basados en diccionarios, ya que la sal limita de forma drástica la velocidad de dichos ataques.

Recientemente, los atacantes han comenzado a explotar las capacidades masivamente paralelas de las tarjetas gráficas (GPU) modernas e, incluso, hardware específico a medida para acelerar los ataques por fuerza bruta a niveles sin precedentes, probando todas las contraseñas posibles de cierta longitud en muy poco tiempo. Por esta razón, se ha hecho necesario usar funciones de derivación de claves basadas en contraseñas (PBKDF), que proporcionan parámetros para ajustar la complejidad temporal y espacial asociada al cálculo del resumen (véase [6]) y, por tanto, reducir la eficacia de los ataques de fuerza bruta basados que empleen este tipo de hardware especializado. Las PBKDF son un área de investigación muy activa, con muchas propuestas recientes (véanse [7], [8], [9], [10], [11], [12] y [13]) que mejoran el estándar actual PBKDF2 (véase [14]).

Además de en la gestión de contraseñas y la derivación de claves, las funciones PBKDF han encontrado aplicaciones en el campo de las criptomonedas y los algoritmos *blockchain*, donde se utilizan como funciones de prueba de trabajo en ciertos diseños (véase [15]).

Por otra parte, el cifrado simétrico (véase [16]) es un tipo de criptografía que emplea las mismas claves para el cifrado y descifrado (o un par de claves con la propiedad de que una sea fácilmente derivable de la otra). Hay dos tipos básicos de criptosistemas simétricos: cifrado en bloque y cifrados en flujo; difieren en que los cifradores en bloque no tienen estado interno y generalmente procesan datos en bloques, mientras que los cifradores en flujo sí tienen un estado interno y procesan datos elemento por elemento (un elemento suele ser un bit o un byte de datos). Sin embargo, la mayoría de los cifradores en bloque se implementan haciendo uso de modos de operación que los hacen funcionar como cifradores en flujo; tal es el caso en nuestra propuesta, donde empleamos el estándar avanzado de cifrado (AES, véase [17]) en modo contador (CTR) para trabajar como un cifrador en flujo y tomar el papel de un generador de números pseudoaleatorios (PRNG). Es destacable que el uso de AES como PRNG ha sido propuesto por el Instituto Nacional de Estándares y Tecnología de los Estados Unidos (NIST, véase [18]). Además de haber sido probado de forma independiente durante casi dos décadas y ser considerado seguro por la comunidad científica, AES presenta la ventaja de estar

acelerado por hardware en los procesadores modernos más comunes, como los que se encuentran en las computadoras portátiles, de escritorio o servidores que utilizamos hoy en día. Esta ventaja permite defenderse contra ataques de GPU o hardware especializado, equilibrando la dificultad entre quienes intentan descifrar las contraseñas y quienes las protegen.

En este trabajo se propone una PBKDF con tres variantes de diferentes características y niveles de optimización, incluyéndose un análisis de rendimiento y seguridad, así como una comparativa con las dos funciones que conforman el estado del arte, Scrypt (véase [7]) y Argon2 (véase [8]).

La función de gestión de contraseñas propuesta presenta un rendimiento y seguridad muy prometedores y puede ser especialmente útil en aplicaciones donde se requiera autentificación segura mediante contraseñas, derivación de claves de cifrado a partir de contraseñas o como función de prueba de trabajo en esquemas *blockchain*.

1.1 Objetivos

El objetivo principal de este trabajo es diseñar una función de derivación de clave basada en contraseña competitiva con el estado del arte actual, tomando como base las ventajas proporcionadas por el uso de AES en modo CTR y maximizando tanto el rendimiento como la seguridad frente a ataques conocidos.

La consecución de este objetivo general requiere los objetivos específicos propuestos a continuación:

- Adaptar la estructura de un generador pseudoaleatorio genérico, y en consecuencia AES en modo CTR, para que pueda funcionar en el contexto de función de gestión de contraseñas.
- Estudiar qué mecanismos se pueden utilizar para establecer la semilla de dicho generador en base a los datos de entrada del usuario.
- Obtener un diseño inicial a partir de las herramientas obtenidas.
- Analizar el rendimiento y seguridad, para desarrollar una versión optimizada del algoritmo que sea competitiva con estándares de reconocido prestigio, como pueden ser Scrypt o Argon2.
- Describir y documentar con precisión la investigación desarrollada en este trabajo.

1.2 Estructura

El presente trabajo consta fundamentalmente, de seis capítulos. En el siguiente capítulo se introduce la terminología y conceptos básicos relacionados con la investigación desarrollada. En el capítulo 3 se expone el estado del arte en materia de funciones resumen criptográficas y, en especial, en la gestión de contraseñas y la derivación de claves. En el capítulo 4 se describe la propuesta inicial, así como la optimización intermedia; mientras que en el capítulo 5 se propone la versión optimizada definitiva y se incluyen comparativas con estándares reconocidos y un análisis de seguridad de la propuesta. Posteriormente, en el capítulo 6, se exponen las conclusiones y líneas futuras de investigación, para finalizar con los anexos que incluyen los materiales y datos asociados al trabajo de investigación desarrollado.

2 Preliminares

En este capítulo se expone la terminología y definiciones básicas relacionadas con la criptografía y la teoría de la información; se abordan propiedades de seguridad como confidencialidad, integridad y autenticidad de los datos o mensajes; finalmente, se describen criptosistemas simétricos (o de clave secreta) y asimétricos (o de clave pública).

Posteriormente, se detallan las funciones resumen (o *hash*) criptográficas, cuya misión principal es convertir un mensaje de longitud variable en un identificador de longitud fija que actúa como resumen del mensaje. También se describen las aplicaciones de estas funciones en el caso de la firma digital de un documento, en la gestión de contraseñas, en la integridad y autenticación de mensajes, así como otras aplicaciones en seguridad informática.
Se introduce, además, la construcción recursiva Merkle-Damgård, base de la gran mayoría de las funciones hash existentes; se hace un breve recorrido histórico y se incluyen aspectos de seguridad y posibles ataques relativos a las funciones resumen criptográficas.

2.1 Definiciones generales

2.1.1 Criptología

La *criptografía* es el conjunto de técnicas basadas en las matemáticas, así como sus aplicaciones informáticas, con el objetivo de ocultar datos ante observadores no autorizados mediante el uso de algoritmos y al menos una clave. Por lo tanto, la criptografía estudia los métodos de cifrado de un mensaje de forma que sea ininteligible para un atacante, brindando mecanismos matemáticos para cifrar y descifrar la información.

Cuando nace la criptografía, surge también la necesidad de analizar la información protegida para determinar si será posible recuperarla, aunque no se conozca el sistema que la ocultó o la clave de cifrado. A esto, se le denomina *criptoanálisis* y es el estudio de los sistemas criptográficos con el objetivo de comprometer la seguridad estos. El criptoanálisis es la contrapartida de la criptografía.

El objetivo del criptoanálisis es obtener, a partir de un texto cifrado, el texto en claro (o una parte) sin poseer el secreto necesario, es decir la clave. Cuando este objetivo se puede cumplir en un tiempo computacionalmente admisible, se dice que el algoritmo o criptosistema se ha roto (véase [19]). El criptoanálisis es, esencialmente, positivo al encargarse de verificar la seguridad y detectar las posibles falencias que pudieran existir en los algoritmos criptográficos. A quienes trabajan en las tareas del criptoanálisis se les denomina c*riptoanalistas* (véase [2]).

Un *criptosistema* es el conjunto de pasos o procedimientos por el cual se alcanza un mensaje cifrado (o *criptograma*) e incluye los pasos inversos por los cuales se obtiene el mensaje en claro. A veces, recibe nombres como primitiva, cifrador, algoritmo criptográfico y, en algunos casos, simplemente cifra (véase [20]).

La *esteganografía* implica el ocultamiento de información de manera que pase desapercibida en su medio habitual, de modo que no se distinga la presencia de los datos. Trabaja con diferentes técnicas de ocultamiento cuya finalidad es insertar o esconder información sensible en un archivo denominado fichero contenedor (véase [2]). Aunque está relacionada con la seguridad de la información, no se encuentra en ámbito de la criptografía al no emplear ninguna clave de cifrado y basarse su seguridad únicamente en el ocultamiento.

La *criptología* es la ciencia que comprende conjuntamente las ramas de la criptografía y del criptoanálisis (véase [21]). Se fundamenta en las matemáticas, y en particular en la matemática discreta. La edad moderna de la criptografía comenzó a mediados del siglo XX, desarrollando nuevas y mejores técnicas basadas en la teoría de la información, la teoría de números y la teoría de la complejidad algorítmica.

La *clave* (denominada también llave), es la pieza de información que aplicada al algoritmo permite transformar el mensaje en criptograma y viceversa. Al conjunto de todas las claves posibles, se le conoce como *espacio de claves*. Resulta esencial que este espacio de claves sea lo suficientemente grande para que un atacante no pueda romper el cifrado simplemente probando todas las claves, o lo que se conoce como *ataque por fuerza bruta* o *búsqueda exhaustiva*. Al conjunto de mensajes se denomina *espacio de mensajes*.

Para cumplir las funciones requeridas en el envío de mensajes usando criptografía, es necesaria la descripción de sus componentes: *transmisor* es quien realiza el proceso de cifrado, *receptor* es quien realiza el proceso de descifrado, *canal* es el medio utilizado para intercambiar la información y *protocolo* es el conjunto de reglas que permiten intercambiar información entre entidades. El *sistema de gestión de claves* es el método para administrar el conjunto completo de claves de las entidades del sistema.

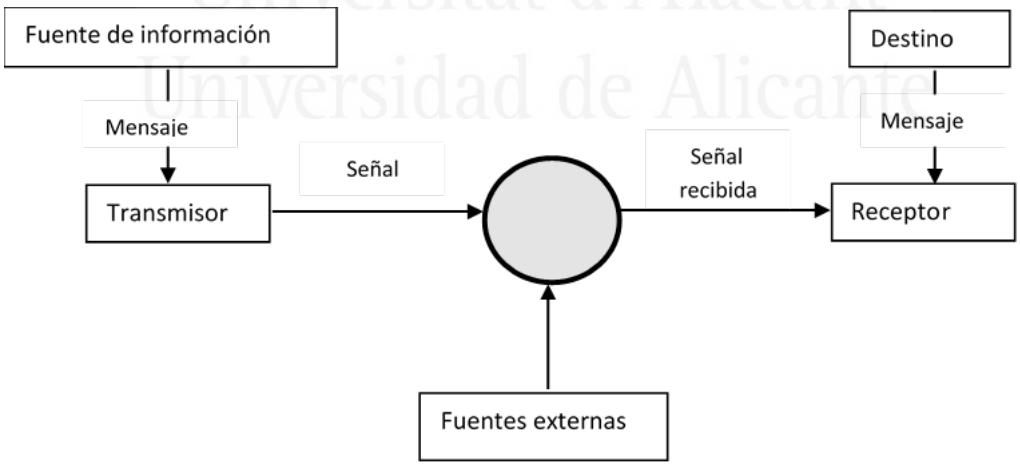


Figura 1. Esquema de comunicación de Shannon

La actual teoría criptográfica se basa en el esquema de comunicación (Figura 1) descrito por Claude E. Shannon a finales de los años cuarenta (véase [19], [22]).

2.1.2 Teoría de la información

En el rendimiento de cualquier sistema de comunicación intervienen los factores de potencia de señal, ancho de banda y la presencia de ruido de fondo no deseado. Basándose en estos conceptos, Claude Shannon desarrolló una teoría de la comunicación completamente nueva en 1948 (véase [22]), a la que llamó teoría matemática de la comunicación, en la que estableció los límites teóricos básicos sobre los rendimientos de los sistemas de comunicación y puso de manifiesto que el mensaje debería ser representado por su información en lugar de por la señal. Su investigación innovadora pronto fue rebautizada como teoría de la información, un área que combina matemáticas e ingeniería y se ocupa tanto de la protección contra el ruido como del uso eficiente del canal.

La teoría de la información se ocupa de tres conceptos básicos:

- La velocidad a la que la fuente genera la información o la *medida de la información de origen*.
- La velocidad máxima a la que es posible la transmisión de información de forma robusta a través de un canal determinado o la *capacidad de información de un canal*.
- La utilización eficiente de la capacidad del canal para la transferencia de información o *codificación*.

Estos tres conceptos están vinculados a través de una serie de teoremas que forman la base de la teoría de la información. Si la tasa de información de una fuente no excede la capacidad del canal de comunicación, existe una técnica de codificación que permita enviar la información a través del canal con una frecuencia de errores arbitrariamente pequeña, a pesar de la presencia de ruido. De esta forma, se puede garantizar la transmisión sin errores a través de un canal de comunicación ruidoso con la ayuda de la codificación; que, cuando es óptima, proporciona un canal sin ruido equivalente con una capacidad de transmisión de información bien definida.

En un sistema de comunicación, la información es algo que se transmite desde la fuente al destino y que este desconocía con anterioridad. Cualquier fuente de información, ya sea

analógica o digital, produce un mensaje (o evento), cuyo resultado se selecciona al azar de acuerdo con una distribución de probabilidad. Un mensaje que contiene información con una probabilidad de ocurrencia baja transmite más información que uno con una alta probabilidad de ocurrencia. La entropía de la fuente es la información promedio por mensaje individual emitido por dicha fuente, entendiéndose como promedio estadístico y no aritmético (véanse [23] y [24]).

2.1.3 Teoría de números

La matemática discreta es el estudio de estructuras matemáticas que son fundamentalmente discretas en lugar de continuas. Los objetos discretos a menudo pueden ser enumerados por números enteros.

La teoría de números es una rama de la matemática discreta dedicada principalmente al estudio de los enteros, los números primos y las propiedades de los objetos hechos de enteros (por ejemplo, números racionales) o definidos como generalizaciones de los enteros (por ejemplo, enteros algebraicos).

La relación con la criptografía y la seguridad de la información es extensa, el algoritmo RSA basa su seguridad en la dificultad de factorizar un número muy grande en factores primos, El-Gamal y Diffie-Hellman se basan en el problema del logaritmo discreto, el criptosistema de McEliece se basa en teoría de códigos, muchos algoritmos de cifrado simétrico se basan en operaciones sobre cuerpos finitos, etc. (véase [25]).

2.1.4 Complejidad algorítmica

Además de la teoría de la información y de la teoría de números, la complejidad algorítmica es otro pilar importante en la criptografía moderna, que se centra en establecer la cantidad de recursos computacionales que requiere un algoritmo para resolver un problema, pudiendo determinar así su eficiencia, rendimiento y seguridad.

Se considera que un problema es intrínsecamente difícil si su solución requiere recursos significativos, independientemente del algoritmo utilizado. Para formalizar este concepto, se introducen modelos matemáticos de computación para estudiar un problema y cuantificar su complejidad computacional o, en definitiva, la cantidad de recursos necesarios para resolverlo,

como tiempo y almacenamiento. Una de las funciones de la teoría de la complejidad computacional es determinar los límites prácticos de lo que las computadoras pueden y no pueden hacer.

Los problemas computacionalmente difíciles están íntimamente ligados con el nivel de seguridad de las primitivas criptográficas actuales (véase [26]).

2.2 Métodos criptográficos

Dentro de la criptografía tenemos dos grandes tipos con aplicaciones muy distintas: simétrica y asimétrica.

La criptografía simétrica, o de clave secreta, se llama así porque utiliza la misma clave para cifrar que para descifrar y porque dicha clave debe permanecer en secreto para garantizar la seguridad. Se utiliza para el cifrado en general ya que suele ser la opción más eficiente. Dentro de este tipo de criptografía, tenemos dos clases de algoritmos distintos: cifradores en flujo y cifradores en bloque.

La criptografía asimétrica, o de clave pública, toma su nombre al utilizar dos claves distintas: una para cifrar y otra para descifrar. En general, una se ha de mantener en secreto (la clave privada) y otra se deja a disposición de otros (la clave pública). Este tipo de criptografía suele hacer uso de funciones matemáticas muy complejas y es mucho más lenta que la criptografía simétrica; por eso, no se utiliza para el cifrado en general. No obstante, permite una serie de aplicaciones que no son posibles o prácticas con criptografía simétrica, como el intercambio seguro de clave o la firma digital.

2.2.1 Criptografía simétrica

La criptografía simétrica es la herramienta básica de cifrado que se utiliza en la gran mayoría de los casos, ya que es rápida, práctica y segura. Como contrapartida, la criptografía asimétrica es mucho más lenta, pero permite funcionalidades adicionales como la firma digital o el intercambio de claves seguro.

Dentro de la criptografía simétrica existen dos grandes clases de criptosistemas: los cifradores en bloque y los cifradores en flujo. Si bien son dos enfoques o diseños distintos, ambos tienen una funcionalidad, rendimiento y seguridad similar. De hecho, en la práctica, es

muy común convertir los cifradores en bloque en cifradores en flujo empleando modos de operación específicos.

Las principales diferencias entre ambos se describen a continuación.

Los *cifradores en flujo:*

- Toman el mensaje elemento a elemento (bit a bit o byte a byte) y lo procesan a modo de cadena (como si fuera letra a letra).
- Aplican una transformación variable que depende de la clave y de la posición del elemento en el mensaje. De esta forma, el mismo byte (o bit) se cifra de forma distinta en función de su posición.
- Todos los cifradores en flujo actuales se basan en una estructura común conocida como esquema Vernam.

Los *cifradores en bloque:*

- Dividen el mensaje en bloques de varios elementos (128 bits, por ejemplo) y lo cifran bloque a bloque.
- Aplican una transformación fija que depende únicamente de la clave utilizada. Esto significa que, para la misma clave y cifrador, el mismo bloque de texto en claro produce siempre el mismo bloque de texto cifrado.
- Muchos (aunque no todos) se basan en una estructura común conocida como red Feistel.

2.2.1.1 Generadores pseudoaleatorios

Los generadores pseudoaleatorios (o PRNG) son algoritmos que se basan en un esquema muy simple:

- Parten de un valor inicial llamado semilla.
- Mediante una función de evolución toman el valor inicial y obtienen el valor siguiente.
- Progresan de la misma forma para obtener todos los valores de la secuencia, tomando el valor anterior, aplicando la función de evolución y obteniendo el valor siguiente.

De esta forma, la semilla determina la secuencia producida. Para una misma semilla, el PRNG generará siempre la misma secuencia mientras que para semillas distintas generará secuencias distintas.

Todos los PRNG producen secuencias que acaban por repetirse tras una longitud determinada. Esta longitud se denomina período de la secuencia. Es importante evitar que la secuencia se repita (cambiando la semilla antes de que esto ocurra) puesto que dejaría de ser aleatoria e impredecible y, por lo tanto, insegura.

Además, el tamaño de la semilla es un factor importante ya que determina el número de semillas posibles y, en definitiva, el número de semillas que un atacante tendría que probar para encontrar la semilla que produce una secuencia determinada. La semilla es un concepto similar al de clave en un cifrador (y suele coincidir con la clave en un cifrador en flujo) por lo que su tamaño debe ser suficiente (128 o 256 bits) para garantizar la seguridad.

La semilla deberá ser un valor aleatorio y por lo tanto impredecible si queremos que la secuencia generada a partir de la misma también lo sea: de poco valdría tener un PRNG excelente si siempre se utiliza la misma semilla y es conocida por el atacante.

A la hora de elegir un generador pseudoaleatorio se puede tomar diferentes alternativas: emplear algoritmos que han sido diseñados a propósito para ser PRNG, tomar un cifrador en flujo (puesto que son prácticamente lo mismo) o adaptar otras primitivas existentes para que funcionen a modo de PRNG.

Se analizan, a continuación, algunos de los PRNG más populares.

Congruencial lineal

Este algoritmo es inmensamente popular ya que se utiliza por defecto en casi todos los lenguajes de programación. Se basa en la siguiente expresión:

$$X_{n+1} = (a \cdot X_n + c) \bmod m$$

Donde el valor anterior (X_n) se multiplica por un coeficiente (a), se le suma un desplazamiento (c) y todo ello módulo m constituye el siguiente valor de la secuencia (X_{n+1}).

A pesar de generar secuencias que, desde el punto de vista de la aleatoriedad estadística, son bastante buenas y que pueden tener un período bastante largo si se eligen los parámetros de forma adecuada, resulta inadecuado para su uso en criptografía o seguridad de la información ya que es predecible: bastan cuatro valores de la secuencia para poder determinar la semilla y los parámetros necesarios (a, c y m) con los que reproducir toda la secuencia.

Lamentablemente, al ser tan popular, se utiliza con mucha frecuencia por programadores inexpertos en tareas de seguridad, permitiendo ataques relativamente sencillos.

Blum Blum Shub

Este PRNG debe su nombre a sus autores y, al contrario que con el congruencial lineal, es demostrablemente seguro al basarse en problemas matemáticos bien conocidos. Sigue el siguiente algoritmo

$$X_{n+1} = (X_n)^2 \bmod m,$$

en el que el valor anterior se eleva al cuadrado y se aplica el módulo *m* para obtener el valor siguiente y donde *m* es el producto de dos números primos muy grandes (miles de bits):

$$p \equiv q \equiv 3(\mathrm{mod}4)$$

$$m = pq$$

A pesar de su gran seguridad, es extremadamente lento (al utilizar números tan grandes) por lo que no se utiliza en la práctica y solo tendría sentido su aplicación para la generación de secuencias muy cortas.

PRNG basados en otras primitivas

Partiendo de un cifrador en bloque seguro, se puede obtener un PRNG seguro (y bastante eficiente) siguiendo esquemas o métodos bien conocidos como modos de operación especiales o esquemas como el estándar ANSI X9.17. Tratamos los cifradores en bloque más adelante.

También se pueden emplear las funciones hash, MAC o PBKDF a modo de PRNG para obtener secuencias pseudoaleatorias.

Hay que tener en cuenta que los cifradores en flujo son, básicamente, PRNG rápidos y seguros por lo que se puede utilizar cualquier cifrador en flujo como PRNG también.

2.2.1.2 Cifrado en flujo

El esquema de Vernam consiste en utilizar una secuencia aleatoria (llamada secuencia cifrante) de la misma longitud que el mensaje y cifrar mediante la operación XOR (*o* exclusiva bit a bit) entre el mensaje y la secuencia cifrante, obteniendo el texto cifrado.

El XOR posee la propiedad de anularse al ser aplicado dos veces, de forma que descifrar es simplemente cifrar de nuevo el texto cifrado (hacer XOR dos veces con la secuencia cifrante), obteniendo el mensaje original.

Utilizar una secuencia aleatoria tan larga como el mensaje no es viable en la práctica, ya que sería necesario que el emisor y el receptor compartieran dicha secuencia de forma segura; lo que no tiene sentido al ser equivalente a compartir directamente el mensaje de forma segura sin necesidad de criptografía. Por ello, en la práctica, se utiliza un generador pseudoaleatorio para generar la secuencia cifrante, de forma que solamente hace falta compartir la semilla (que actúa como clave en este esquema) y que es, en general, mucho más corta que el mensaje completo.

Para que el esquema Vernam sea seguro, se debe utilizar un generador de secuencia cifrante impredecible y que tenga un período suficientemente largo (cambiando de semilla o clave antes de que se repita dicha secuencia). Además, este generador ha de aceptar semillas suficientemente largas para que el espacio de claves sea seguro ante un ataque por fuerza bruta. En la actualidad, se considera seguro una semilla (o clave) de 128 bits, si bien es ideal utilizar 256 bits para tener un cierto margen de seguridad adicional.

Se analizan, a continuación, los cifradores en flujo más representativos.

LFSR

Los registros de desplazamiento con retroalimentación lineal (o LFSR de sus siglas en inglés) son muy populares para la generación de secuencias pseudoaleatorias y, por tanto, para el diseño de cifradores en flujo.

Estos registros poseen ciertas características deseables:

- Se pueden analizar de forma sencilla dado que tienen propiedades matemáticas conocidas.
- Su implementación en hardware es eficiente y directa (aunque no tanto en software).
- Con ciertas funciones de retroalimentación determinadas, se puede garantizar un cierto período (longitud) de la secuencia antes de que se repita.
- Las secuencias generadas tienen muy buenas propiedades de aleatoriedad estadística.

Si bien estas propiedades son muy positivas, los LFSR por sí solos no son seguros puesto que son predecibles y su uso en criptografía no está muy recomendado. No obstante, puesto que

poseen estas propiedades tan atractivas desde el punto de vista matemático, se han utilizado (y se utilizan) combinados con otras técnicas para intentar explotar sus características de aleatoriedad de forma segura.

RC4

RC4 es un algoritmo de cifrado en flujo tremendamente popular. Diseñado por Ron Rivest en 1987, se mantuvo en secreto hasta que en 1994 se publicó de forma anónima en internet.

Dado que es tremendamente eficiente y sencillo, se ha utilizado para el cifrado en multitud de protocolos de seguridad como SSL/TLS (el protocolo utilizado para conectar a web seguras) o WEP/WPA (protocolos de seguridad para WiFi).

Lamentablemente, recientemente se han publicado ataques cada vez más efectivos contra RC4 generando mucha desconfianza en su diseño, llegando a ser prohibido en las versiones actuales de SSL/TLS.

A pesar de que no hay un ataque definitivo contra RC4, su uso no es recomendable.

Salsa20

Salsa20 es un cifrador en flujo diseñado por Dan Bernstein para el concurso europeo eStream. Está considerado seguro y es muy eficiente en software al basarse únicamente en 3 operaciones: suma de valores de 32 bits, XOR bit a bit y rotaciones de bits.

Tiene como característica interesante que su PRNG es capaz de generar cualquier punto de la secuencia cifrante sin tener que generar los valores anteriores (esto es algo bastante inusual). Existen variantes de menos rondas (más rápidas, pero menos seguras) como Salsa20/12 o Salsa20/8, pero no se recomienda su uso. También existe una versión mejorada que se llama ChaCha aunque no goza de la misma popularidad que Salsa20.

Salsa20 (o su variante ChaCha) es el cifrador en flujo recomendado en la actualidad.

Otros

Si bien la recomendación es utilizar Salsa20 (o ChaCha), hay muchos cifradores en flujo interesantes desde el punto de vista histórico o del diseño, entre otros:

- Los cifradores *A5* (tanto A5/1 como A5/2) basados en LFSRs y utilizados en el estándar de *telefonía GSM*.
- El cifrador en flujo *SEAL*, diseñado en 1997 por IBM y que es bastante rápido en software al basar su diseño en la generación de tablas que dependen de la clave. El hecho de haber sido patentado a afectado negativamente a su popularidad.
- *Phelix* (y su versión original *Helix*) es un cifrador en flujo diseñado por Bruce Schneier y otros para el concurso *eStream*. Muy eficiente en plataformas de 32 bits, ha sufrido ciertos ataques que han impedido su uso generalizado.
- *HC-128* (y su versión mejorada *HC-256*) es un diseño de Hongjun Wu publicado en 2004 y no patentado que también participó en *eStream*. No se conocen ataques y es bastante eficiente, aunque requiere de una inicialización (para cada cambio de clave) muy lenta. No es tan popular como Salsa20.
- *SNOW 3G* es un algoritmo propuesto por Johansson y Ekdahl en 2006 y está basado en un LFSR combinado con una máquina de estados finitos (FSM en inglés). Es el cifrador elegido para el protocolo 3GPP *(telefonía 3G y 4G)*.
- *Spritz* fue propuesto por Rivest (el autor de RC4) en 2014 como sustituto de RC4. Es una modificación extensa de RC4 basada en un diseño de función esponja que, si bien es mucho más seguro, también es bastante más lento por lo que no ha gozado de la misma popularidad que RC4 o Salsa20.

2.2.1.3 Cifrado en bloque

Algunos cifradores en bloque siguen un esquema común que se llama red Feistel, por su autor. Horst Feistel propuso un diseño de cifrador en bloque basado en unos conceptos sencillos:

- Propone una aproximación al cifrador en bloque ideal (que no es realizable en la práctica) mediante un cifrador producto. Esto consiste en ejecutar operaciones más o menos sencillas en secuencia, o por rondas, de forma que se incremente la complejidad al aplicar cada ronda de forma sucesiva.
- También indica la necesidad de que el cifrador cuente con un tamaño de clave (*k* bits) independiente del tamaño del bloque (*n* bits), de forma que el número de transformaciones posibles (o espacio de claves) sea 2^k.
- Por último, basa su diseño en un cifrador que alterna sustitución (cambiar un elemento por otro) y permutación (reordenar elementos).

En general, todos los cifradores en bloque basados en una red Feistel son parecidos, pero se diferencian en lo siguiente:

- El tamaño del bloque, un valor común en la actualidad sería 128 bits.
- El tamaño de la clave, se considera que 128 bits es seguro, pero 256 bits es lo ideal para tener cierto margen de seguridad.
- El algoritmo de generación de subclaves, que se encarga de derivar valores para cada ronda a partir de la clave de cifrado/descifrado. Su complejidad es uno de los factores que determinan la seguridad general del cifrador.
- La función de ronda, que se encarga transformar una de las dos mitades en cada ronda y cuyo diseño es el otro factor principal de la seguridad del cifrador.
- El número de rondas, un número común es 16 pero depende completamente del diseño del resto del cifrador. A más rondas, más complejidad y seguridad, pero también peor rendimiento. Esto lleva a buscar un equilibrio entre seguridad y velocidad de cifrado.

En definitiva, ciertos autores basan sus diseños en una red Feistel puesto que es sencilla de comprender y aporta ciertas garantías como que el cifrador se pueda descifrar y que sea seguro si la función de ronda elegida también lo es.

Entre los cifradores en bloque más populares se encuentran DES y AES, así como otros relevantes por sus características específicas.

DES

Horst Feistel dirigió un proyecto en IBM a finales de los años 60 que dio como resultado un cifrador llamado *Lucifer*. Este algoritmo fue uno de los candidatos del concurso *DES* (Data Encryption Standard o estándar de cifrado de datos) y cuando ganó se convirtió en DES, el primer estándar de cifrado de datos moderno.

Las diferencias entre DES y Lucifer no son simplemente un cambio de nombre, la NSA modificó el diseño para mejorar la seguridad y redujo la clave de 128 a 56 bits para que cupiera en un único chip (el hardware de la época era mucho más limitado que el actual).

Estas modificaciones provocaron el recelo de la comunidad científica, ya que se había reducido la clave de forma significativa y los criterios que habían motivado los cambios sobre el diseño de Lucifer no se hicieron públicos. Más tarde, se descubrió que la NSA conocía entonces

un ataque (criptoanálisis diferencial) que la comunidad científica no descubrió hasta más tarde y modificó Lucifer para hacerlo más resistente.

No obstante, la clave de 56 bits que utiliza DES no se debería considerar segura en la actualidad ya que es demasiado fácil (o rápido) de romper por fuerza bruta con la capacidad computacional que tenemos hoy en día (ordenadores mucho más rápidos y baratos, computación en la nube, etc.).

Si bien DES ha gozado de gran popularidad, no tiene sentido utilizarlo en la actualidad: es un diseño arcaico, ineficiente en software y con una longitud de clave insuficiente. Es posible que en algunos sectores se utilicen aplicaciones o sistemas antiguos heredados que todavía hagan uso de chips DES para cifrar datos. En estos casos y para que sea seguro, se utiliza de una forma especial que se llama triple DES y que supone hacer tres cifrados DES (en realidad dos cifrados y un descifrado) y duplicar la clave de 56 a 112 bits (que se podría considerar seguro hoy en día). A pesar de triple DES, su uso no es recomendable para ninguna aplicación actual.

AES

Una vez se puso de manifiesto que DES estaba quedando desfasado muy rápidamente, el *NIST* (instituto de estandarización y tecnología de EE. UU.) convocó otro concurso para establecer un nuevo estándar de cifrado más avanzado. Lo que dio lugar al concurso *AES* (Advanced Encryption Standard o estándar de cifrado avanzado) en el año *2001,* que ganó un algoritmo originalmente llamado *Rijndael* (por sus autores belgas, Joan Daemen y Vincent Rijmen).

Este nuevo estándar, AES, tiene ciertas peculiaridades interesantes: *no utiliza una red Feistel*, se basa en *operaciones matemáticas relativamente complejas* (suma, producto y división sobre un cuerpo de Galois) y está diseñado para ser *implementado de forma eficiente en software* (tanto en máquinas de 8 bits como en procesadores más complejos de 32 bits). Además, la gran mayoría de procesadores actuales soportan *aceleración por hardware* para AES, mejorando el rendimiento de forma muy significativa y poniéndolo en el mismo nivel que cifradores en flujo muy rápidos como RC4 o Salsa20.

A pesar de no ser estrictamente una red Feistel, sí que se basa en conceptos similares: varias rondas idénticas que se parametrizan con una subclave distinta para cada ronda, etc. En AES el bloque se considera como una matriz de 4x4 bytes (16 bytes o 128 bits) y las rondas

comprenden 4 operaciones básicas todas utilizando esta aritmética especial sobre un cuerpo de Galois: sustituir elementos mediante una tabla (*SubBytes*), reordenar filas (*ShiftRows*), producto matriz vector (*MixColumns*) y combinar la subclave de ronda con el bloque (*AddRoundKey*).

AES permite claves de 128, 192 o 256 bits simplemente incrementando el número de rondas para los tamaños de clave mayores y es considerado seguro en la actualidad. Dado que está acelerado por hardware en muchos sistemas, es una opción idónea para el cifrado de propósito general y es el cifrador en bloque recomendado en la actualidad.

Otros cifradores en bloque

Si bien nuestra recomendación es utilizar AES (idealmente con clave de 256 bits), hay muchos cifradores en bloque interesantes desde el punto de vista histórico o del diseño, entre otros:

- El cifrador IDEA, un estándar internacional de cifrado en bloque publicado en 1991 y que no ha gozado de la popularidad de DES o AES.
- Los algoritmos de Ron Rivest RC5 y RC6 que, al contrario que RC4, son cifradores en bloque, pero no han tenido éxito en la práctica.

El algoritmo Blowfish (Schneier y otros en 1993, véase [27]) y su versión mejorada Twofish (1998, véase [28]) que tienen un diseño interesante y han gozado de cierta popularidad en el mundo del software libre y la gestión de contraseñas.

2.2.1.4 Modos de operación en cifrado en bloque

Los cifradores en bloque no se suelen utilizar directamente siguiendo su diseño básico, se aplican en distintos modos de operación que, partiendo del cifrador en bloque básico, obtienen mejoras significativas en ciertos aspectos. Los modos más comunes son los siguientes:

Cifrado triple. Este modo se utiliza en DES para aumentar la seguridad y duplicar el tamaño de la clave. Se realiza un cifrado con una clave K1, un descifrado con una clave K2 y otra vez un cifrado con la clave K1; de esta forma se tiene un rendimiento 3 veces peor, pero se utilizan dos claves distintas y, por lo tanto, se duplica el espacio de claves y la seguridad frente a un ataque por fuerza bruta. Se podría aplicar a cualquier cifrador, pero sólo tiene sentido realmente en el caso de DES ya que su clave es de 56 bits y no es seguro sin aplicar el cifrado triple, no recomendándose su utilización en otro caso.

Modo ECB (libro de código electrónico). Este es el modo directo de utilizar un cifrador en bloque, sin aplicar ningún modo de operación avanzado. Presenta el gran problema de que, para la misma clave, la misma entrada siempre produce la misma salida; lo que implica que, muchas veces, no esconde los patrones estadísticos o de frecuencia de la información original. Además, es necesario conformar el texto en claro al tamaño del bloque, esperando a tener información suficiente para rellenar un bloque completo o rellenar el último bloque si el mensaje no tiene una longitud múltiplo del tamaño del bloque. Por estas razones, no se recomienda utilizar el modo ECB.

Modo CBC (encadenamiento de bloque cifrado). En este modo se utiliza el bloque cifrado anterior para hacer el XOR con la entrada del bloque siguiente; como en el caso del primer bloque no hay bloque anterior, se utiliza un valor inicial aleatorio (IV) que no tiene porqué ser secreto, pero debe estar disponible para el descifrado. De esta forma se evita que la misma entrada, con la misma clave, produzca siempre la misma salida. No obstante, sigue presentando el inconveniente de tener que conformar el mensaje al tamaño del bloque y, además, este encadenamiento fuerza a una ejecución secuencial (no permite paralelismo o ejecución sobre múltiples núcleos).

Modo CFB (realimentación de cifrado). Este modo convierte el cifrador en bloque en un generador pseudoaleatorio que se utiliza a modo de cifrador en flujo. Para ello, encadena la salida cifrada como entrada del siguiente bloque y genera un bloque cifrado que utiliza como secuencia para cifrar mediante XOR (al estilo Vernam) con el texto en claro. Esto mejora el modo CBC ya que no requiere conformar el mensaje al tamaño del bloque, pero tiene penalizaciones en el rendimiento.

Modo OFB (realimentación de salida). Es similar al anterior, pero realimenta la salida del cifrador en bloque en lugar de la salida del cifrado final. Es muy parecido a un cifrador en flujo, pero con elementos de tamaño bloque en lugar de bit o byte. Como el encadenamiento no utiliza el texto en claro, permite realizar el cifrado de varios bloques mientras se espera la recepción de todo el mensaje.

Modo CTR (contador). Este modo no realiza encadenamiento, emplea un contador cuyo valor se va incrementando de uno en uno y utiliza la salida del cifrado en bloque como secuencia cifrante. El valor inicial del contador no debe repetirse, pero puede ser público (el secreto es la clave). Al no realizar encadenamiento, permite su ejecución en paralelo. El modo CTR sería la

recomendación tanto por seguridad y eficiencia en la mayoría de los casos (salvo que se necesite alguna característica concreta de los modos anteriores).

Modo XTS (cifrado de disco). Este modo está indicado únicamente para el cifrado de discos ya que tienen características especiales. Utiliza dos claves y varios cifrados y tiene en cuenta la posición de los datos en el disco para el cifrado. La mayoría de los sistemas de cifrado de disco (Bitlocker de Microsoft o FileVault de Apple, etc.) utilizan el modo XTS con AES; aunque no son necesariamente compatibles entre sí.

2.2.2 Criptografía asimétrica

La criptografía de clave pública o asimétrica se diferencia de la tradicional criptografía simétrica (cifrado en bloque o en flujo) en que utiliza dos claves asociadas, una para cifrar y otra distinta para descifrar. Esta característica permite funcionalidades adicionales que son imposibles, o muy difíciles, en la criptografía simétrica: la distribución de claves mediante un canal inseguro y la firma digital.

En la criptografía asimétrica, cada usuario posee un par de claves: una clave pública y otra privada. Como su nombre indica, la clave pública se puede transmitir e incorporar en repositorios públicos, pero la privada ha de permanecer en secreto. Si se cifra con la clave pública, se podrá descifrar con la privada y al revés, si se cifra con la clave privada, se podrá descifrar con la pública.

Las tres operaciones básicas que permite un criptosistema de clave pública son:

- *Cifrado*. Para enviar un mensaje cifrado de Alicia a Bernardo (de A a B), Alicia cifra el mensaje con la clave pública de Bernardo; luego, Bernardo puede descifrar el mensaje haciendo uso de su clave privada. Como sólo Bernardo tiene su clave privada, sólo Bernardo puede descifrar el mensaje. Manolo, que es un espía, sólo puede ver el mensaje cifrado y la clave pública de Bernardo, pero no tiene la clave privada para descifrar el mensaje. Para cifrar un mensaje, se cifra con la clave pública del destino y se descifra con la clave privada del destino.

- *Firma*. Para firmar un mensaje, Alicia cifra el mensaje con su clave privada y lo envía a Bernardo; luego, Bernardo puede comprobar que el mensaje proviene de Alicia descifrando el mensaje con la clave pública de Alicia y, comprobando que coincide con el mensaje original. Como sólo Alicia tiene su clave privada, ella es la única capaz de firmar un mensaje que se descifre correctamente con su clave pública y, por lo tanto, queda garantizada su autoría. El espía Manolo también puede comprobar la firma de Alicia, puesto que tiene su clave pública, pero no puede firmar ningún mensaje en nombre de Alicia. Es importante destacar que la firma digital garantiza la autenticidad o identidad de los datos, pero no la confidencialidad; si queremos que el mensaje sea privado, debemos cifrar tras firmar el mensaje. Para firmar un mensaje, se cifra con la clave privada del firmante y se comprueba dicha firma descifrando con la clave pública del firmante.
- *Intercambio de clave*. En muchos casos, nos basta con intercambiar una clave de forma segura entre Alicia y Bernardo para después utilizar un cifrador convencional (AES o Salsa20, por ejemplo) para cifrar los datos; esto es lo que realizan protocolos como TLS o SSH. En clave pública, Alicia puede intercambiar una clave con Bernardo simplemente generándola de forma aleatoria y cifrándola con la clave pública de Bernardo. No obstante, hay algoritmos de clave pública que no permiten ni cifrar ni firmar, pero sí compartir de forma segura un mismo número en ambos extremos a partir del cual se puede derivar una clave a utilizar para criptografía simétrica.

Los conceptos importantes a tener en cuenta son:

- *Rendimiento*. La criptografía asimétrica es demasiado lenta para el cifrado de datos en general, por lo que recurrimos a un intercambio de clave y al cifrado mediante criptografía simétrica del resto de los datos en la mayoría de los casos. Lo mismo ocurre con la firma digital, que es demasiado lenta para firmar mensajes muy largos; aquí recurrimos a calcular el resumen del mensaje mediante una función hash y firmar el resumen en lugar del mensaje completo por motivos de eficiencia.
- *Seguridad*. La criptografía asimétrica no es más segura que la criptografía simétrica, depende del tamaño de la clave y del coste computacional asociado a probar todas las claves posibles. Hay que utilizar tamaños de clave tanto para la criptografía asimétrica como para la simétrica que sean equivalentes en cuanto a seguridad.

- *Distribución de claves*. La criptografía de clave pública requiere de ciertas infraestructuras y protocolos para la gestión de las claves: repositorios de claves públicas, terceras partes de confianza que garanticen la autenticidad de las claves, etc. Sin estas medidas, el espía Manolo puede realizar lo que se conoce como ataque del hombre en el medio, haciéndose pasar por Bernardo frente a Alicia y como a Alicia frente a Bernardo y filtrando toda la información que, en teoría, debería ser privada entre ambos.

A continuación, se describen los algoritmos principales que existen para realizar criptografía asimétrica o de clave pública.

RSA

Se propone en 1978 y es el primer algoritmo de clave pública completo en salir al mercado. Proviene de Rivest (creador de RC4, MD5, etc.), Shamir y Adleman. Su seguridad reside en el problema de la factorización de un número en factores primos cuando este número es muy grande. El algoritmo RSA ha estado patentado hasta el año 2000, por lo que era necesario pagar para su utilización.

Permite tanto el cifrado (y por tanto el intercambio de clave) como la firma digital y está considerado como seguro en la actualidad. Si bien, dejaría de ser seguro (al igual que los otros criptosistemas de clave pública) si tuviéramos computadores cuánticos viables. Los computadores cuánticos podrían realizar los cálculos asociados a romper los algoritmos de clave pública de forma mucho más eficiente que los computadores convencionales. Esto ha dado lugar al concepto de criptografía postcuántica, en donde se intenta diseñar algoritmos capaces de resistir un ataque mediante un computador cuántico.

RSA es extremadamente popular y se utiliza en multitud de aplicaciones. Se recomienda su uso con una clave de 3072 bits (equivalente a 128 bits en criptografía simétrica) como mínimo, si bien pueden existir alternativas más eficientes para ciertas aplicaciones.

Diffie-Hellman

El algoritmo Diffie-Hellman sólo permite el intercambio de claves, no permite ni el cifrado ni la firma digital. Se basa en la dificultad de calcular el logaritmo discreto (problema matemático similar al de RSA) y permite llegar al mismo número en ambos extremos de la comunicación.

Este número, que se comparte de forma segura, puede ser utilizado para derivar una clave que se utilizará en criptografía simétrica convencional y depende de las claves públicas y privadas de cada extremo. De esta forma, si queremos compartir una nueva clave, tendremos que generar nuevas claves públicas y privadas.

A pesar de ser exclusivamente un intercambio de clave, ha gozado de mucha popularidad ya que es más rápido que RSA en este cometido y no ha estado patentado, por lo que no era necesario pagar por su uso. Se utiliza en muchos protocolos actuales como TLS o el de cifrado de Whatsapp.

Requiere un tamaño de clave similar a RSA para ser seguro (3072 bits).

Curvas elípticas

La criptografía sobre curvas elípticas es, esencialmente, una mejora sobre Diffie-Hellman utilizando una aritmética distinta que permite claves mucho más pequeñas y rendimientos mucho mejores.

A esta técnica se le denomina ECDH (Elliptic Curve Diffie-Hellman) y es la forma que se utiliza tanto en TLS como Whatsapp. Se recomienda el uso de ECDH con un tamaño de clave de 256 bits como mínimo dado su mayor rendimiento respecto a RSA o Diffie-Hellman convencional.

ElGamal

El algoritmo de Taher Elgamal es una variante sobre Diffie Hellman que permite el cifrado y la firma digital. Se ha utilizado en muchos protocolos (como el estándar DSS de firma digital del NIST o el DSA) para implementar firma digital sin tener que licenciar la patente de RSA. Elgamal fue también uno de los creadores de SSL en Netscape.

2.3 Funciones hash

Una primitiva criptográfica es la combinación de algoritmos, elementos y métodos que forman un criptosistema o un protocolo de seguridad e incluyen comúnmente esquemas y algoritmos de cifrado y de firma digital, así como funciones hash. Se toman como bloques básicos para el diseño, por lo que deben ser muy robustos (véase [2])

Una de las primitivas con más interés en criptografía son las denominadas funciones unidireccionales. Estas son la base de otras funciones criptográficas, como las funciones hash. Estas consisten en una función computable que obtiene, a partir de un mensaje *M* de tamaño variable, una representación de tamaño fijo del propio mensaje, *H(M)*, que se conoce como su resumen o hash. Este valor es, en definitiva, una representación compacta del mensaje de entrada.

El termino hash proviene del significado de los verbos del inglés cortar y mezclar, ya que las funciones hash *cortan* y *mezclan* la entrada para obtener la salida. El primero en aplicar el término fue H. Luhn, de IBM, en 1953, aunque no comenzó a utilizarse masivamente hasta la década del 60, luego de que R. Morris lo aplicara en una publicación técnica (véase [29]).

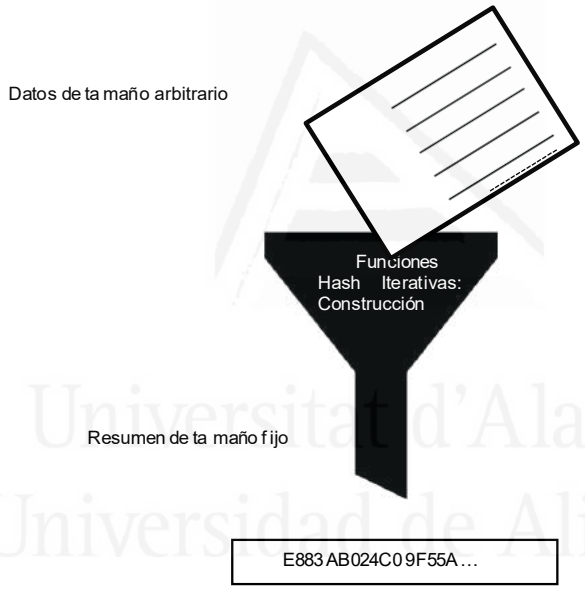


Figura 2. Funcionamiento de una función hash

Una función hash unidireccional es una función hash *H* que cumple que para cualquier resumen *H(m)* es difícil encontrar el mensaje m que produzca dicho resumen. Una buena función resumen es una que tiene pocas colisiones (véase página 30) en el conjunto esperado de entrada.

Entre las propiedades de las funciones hash, se encuentran las siguientes:

- *Determinista*. Un mensaje determinado siempre tiene el mismo valor hash.

- *Distribución uniforme*. Cada valor hash debe tener una probabilidad muy similar de ser generado.
- *Imagen*. El conjunto de posibles valores de salida debe ser finito y bien definido.
- *No invertible*. Las funciones hash deben ser no invertibles o de un solo sentido.
- *Rendimiento*. El valor de salida debe poder ser calculado de una manera eficiente.

2.3.1 Funciones hash criptográficas

Se llaman funciones hash criptográficas a las funciones hash que cumplen los requisitos de seguridad para ser empleadas en criptografía. Este tipo de funciones se caracterizan por presentar propiedades adicionales que las hacen resistentes frente a los ataques que intentan romper la seguridad de los sistemas informáticos.

Una función hash será segura y, por lo tanto, considerada como criptográfica si tiene las siguientes propiedades:

- *Unidireccionalidad*. Cuando computacionalmente es imposible encontrar el mensaje *M*, a partir de su resumen *H(M)*.
- *Avalancha*. Para dos valores de entrada muy similares, los valores hash de salida deben ser diferentes. Se considera que para el cambio de un único bit en un mensaje *M*, el resumen *H(M)* debería cambiar la mitad de sus bits aproximadamente.
- *Resistencia a colisiones*. Hay una colisión (véase página 30) cuando se tienen dos mensajes *M* y *M'* de forma que ambos producen el mismo resumen: *H(M)=H(M')*. Por ejemplo, un algoritmo que devuelve un hash de 128 bits; para que cada hash equivalga a un único mensaje, tendrían que existir solamente 2^{128} textos distintos, lo cual no es posible. Como textos distintos hay infinitos, podemos decir que hay infinitas posibilidades que dos textos tengan el mismo hash o que colisionen. Por tanto, uno de los requisitos de las funciones hash criptográficas es que encontrar una colisión sea lo menos probable posible.

2.3.2 Construcción Merkle-Damgård

Una función hash (o función resumen) criptográfica toma una entrada de tamaño variable y genera una salida de tamaño fijo (resumen). En realidad, las funciones hash tienen una entrada

de tamaño fijo también, pero dividen el mensaje en bloques de ese tamaño y los van procesando iterativamente (uno tras otro), generando un único resumen para todo el mensaje.

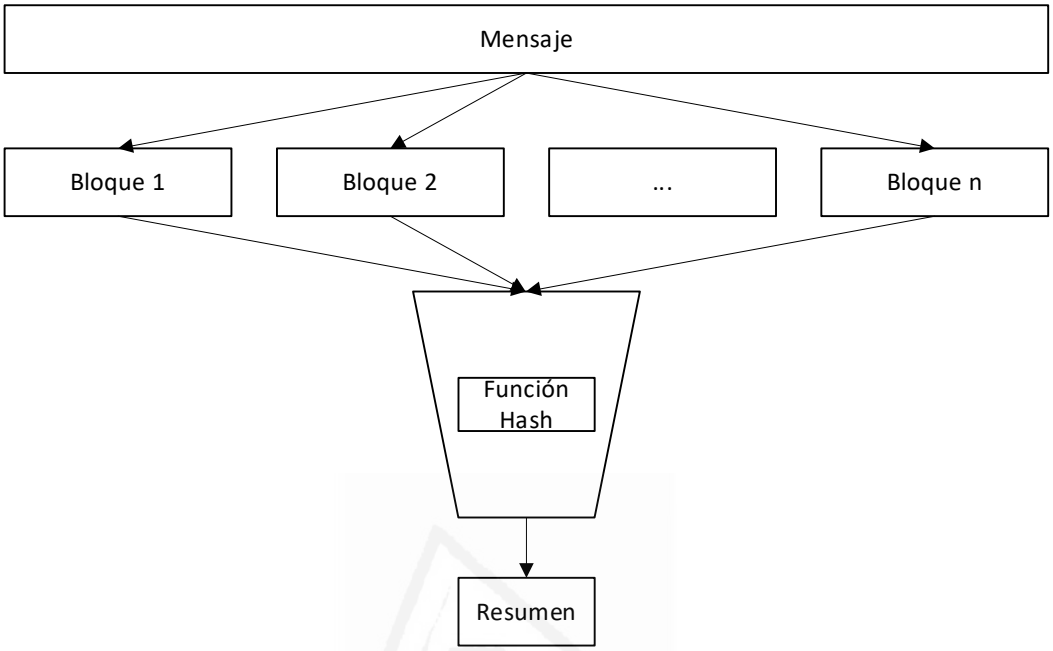


Figura 3. Procesamiento iterativo en una función hash

Como el mensaje no suele ser de una longitud múltiplo del tamaño de entrada de la función resumen, se hace necesario rellenar el último bloque y la mayoría de las funciones hash usan un esquema conocido como Merkle-Damgärd para esto.

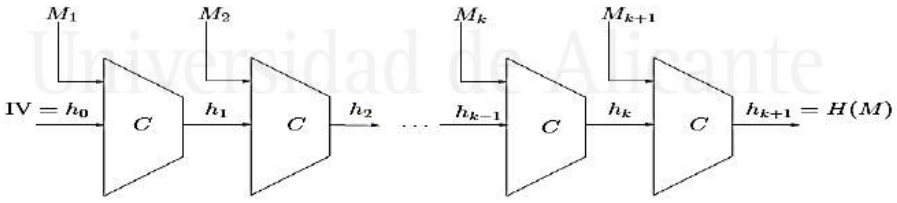


Figura 4. Construcción Merkle-Damgård

Como se puede en la Figura 4, hay dos características a considerar en el diseño de un hash iterativo:

- *El sistema de relleno (o padding)*. Evita colisiones, impidiendo que existan dos mensajes que al rellenarse generen el mismo mensaje de entrada. Su construcción debe ser cuidadosamente escogida para garantizar la seguridad del esquema. Además, debe

generar una entrada cuyo tamaño sea múltiplo de un número fijo, pues la función de compresión *C* no acepta entradas de tamaño arbitrario.

- *La función de compresión.* Combina cada bloque del mensaje con la salida de la ronda anterior, transformando dos entradas de longitud fija en una salida del mismo tamaño que una de las entradas, se inicia con un valor fijo inicial (*IV*) que es parte del diseño de la función hash. La función de compresión es una función de una vía o unidireccional (véase [30]).

De las funciones que se construyen mediante el anterior sistema se dice que son funciones hash iterativas y a la forma de construcción recursiva se la conoce como de Merkle-Damgård (Figura 4) debido a que fue usada por primera vez por R. Merkle y I. Damgård (véase [31]). Sirve para construir funciones hash criptográficas resistentes a colisiones y en ella se basan la gran mayoría de funciones hash disponibles (MD5, SHA-1 y SHA-2 entre otras).

Si la función de compresión es resistente a las colisiones, entonces la construcción resultante también lo será, pero esta construcción es una razón fundamental de que sea erróneo pensar que las funciones hash son cajas negras. La construcción iterativa fue diseñada con el propósito de extender el dominio de las funciones resistentes a las colisiones por lo que no se debe esperar que ofrezcan garantías de seguridad más lejos de su extensión (véase [32]).

2.3.3 Ataques a funciones hash

Colisiones

Como se ha indicado previamente, podemos definir una colisión como la situación en la que dos entradas diferentes a un algoritmo generan la misma salida. En el caso de las funciones hash, esto ocurre cuando dos valores de entrada diferentes generan un mismo resumen, por tanto, una propiedad fundamental es que la función hash debe ser resistente a colisiones.

Una función hash es resistente a colisiones si no es computacionalmente factible hallar un par de mensajes distintos *m*, *m'*, tal que *h(m) = h(m')*. La complejidad de este ataque contra un hash de *n* bits es $O(2^{n/2})$, según la *paradoja del cumpleaños* (véase [33]).

Este concepto se puede extender a dos variantes: encontrar dos mensajes cualesquiera que tengan el mismo resumen (lo que se conoce como *colisión débil*) o, partiendo de un mensaje determinado, encontrar otro mensaje con el mismo resumen (*colisión fuerte*).

Preimagen

Como ya se ha descrito con anterioridad con el concepto de unidireccionalidad (véase página 28), la función hash debe ser resistente a preimagen, es decir, debe ser computacionalmente inviable para un atacante, partiendo de un resumen o hash determinado, generar un mensaje que produzca dicho resumen.

2.3.4 Algoritmos hash comunes

Los hashes criptográficos cuentan con una serie de fortalezas que los hace útiles para su uso en seguridad. En general, cuando se habla de algoritmos de hash en seguridad se hace referencia a estos.

2.3.4.1 Message Digest (MD)

La serie Message Digest (MD) es una familia de funciones hash creada por Ron Rivest, criptógrafo y profesor en el MIT (Instituto Tecnológico de Massachusett) desde el año 1974. Es miembro del Laboratorio de Informática e Inteligencia Artificial (CSAIL) y fundador del Grupo de Criptografía y Seguridad de la Información del MIT. Junto con Adi Shamir y Len Adleman fueron los inventores del algoritmo RSA (véase [34]). Además, Rivest es el inventor de los algoritmos de criptografía simétrica RC2, RC4, RC5, y coinventor de RC6.

También autor de funciones hash criptográficas entre las cuales tenemos la familia MD como: MD2, MD4, MD5 y MD6. En 2006, publicó un sistema de votación auditable ThreeBallot (véase [35]), el cual incorpora la privacidad de voto y la posibilidad de que el votante compruebe si el voto fue contabilizado.

MD2

El algoritmo Message Digest 2 es una de la primeras funciones hash criptográficas desarrollada en 1988 por R. Rivest, publicada en 1989. Fue un algoritmo dirigido para computadores de 8 bits. Su valor hash para entradas de cualquier longitud se forma haciendo que el mensaje sea múltiplo de la longitud de bloque, computable a 128 bits o 16 bytes y añadiéndole una suma de verificación (checksum) para evitar discrepancias en los valores resultantes del mensaje (véase [21]). Ha sufrido numerosos ataques y no puede considerarse seguro (véase [36]).

MD4

Es un algoritmo diseñado por el profesor R. Rivest con un resumen de 128 bits de longitud. La arquitectura del algoritmo ha influido en los diseños siguientes de los algoritmos como MD5, SHA o el RIPEMD-160 (véase [37]). Aunque ciertas debilidades en MD4 fueron demostradas por Den Boer y Bosselaers ya en 1991 (véase [38]), muchos de los diseños posteriores basados en MD4 siguen siendo seguros (SHA-2 o RIPEMD-160), puesto que no se ha publicado ningún ataque eficaz contra ellos (véase [21]).

MD5

El algoritmo MD5, creado por R Rivest en 1992, es una función hash de 128 bits, que toma una entrada un tamaño arbitrario y que genera una salida de 128 bits. Si bien fue muy utilizado, está roto de forma definitiva desde el año 2005. Su diseño fue la mejora de sus antecesores MD2 y MD4.

Su funcionamiento consiste en fraccionar el mensaje que va a ser procesado en bloques de 512 bits de acuerdo con el esquema Merkle-Damgård: en el caso de que el último bloque o el mensaje en sí, sea menor a 512 bits, este se rellena hasta alcanzar 512 bits con un esquema predefinido que incluye la longitud del mensaje. Además, se tiene un búfer estado de 128 bits manejado como cuatro palabras de 32 bits. La función de compresión se efectúa en cuatro rondas, combinando el bloque de 512 bits del mensaje con el búfer de estado, así que cada palabra del mensaje es usada cuatro veces. Después de las cuatro rondas de la función de compresión, el búfer de estado y el resultado son sumados para obtener la salida (véase [16]).

A pesar de haber sido considerado criptográficamente seguro en un principio, ciertas investigaciones han revelado vulnerabilidades que hacen cuestionable el uso futuro del MD5. En agosto de 2004, anunciaron que existían colisiones en la función hash criptográfica MD5 (véase [39]). Aunque dicho ataque era analítico, el tamaño del hash de 128 bits es lo suficientemente pequeño como para que se suponga una vulnerabilidad muy grande frente a ataques de fuerza bruta. El proyecto de computación distribuida MD5CRK arrancó en marzo de 2004 con el propósito de demostrar que MD5 es inseguro frente a uno de tales ataques, aunque acabó poco después de la publicación de la vulnerabilidad definitiva.

Debido al descubrimiento de métodos sencillos para generar colisiones de hash, e incluso ataques de preimagen, muchos investigadores recomiendan su sustitución por algoritmos más seguros tales como SHA-2 o SHA-3 (véase [40]).

2.3.4.2 Secure Hash Algorithm (SHA)

SHA-1

En 1995, el *NIST* estadounidense decide crear el estándar *SHA-1* (Secure Hash Algorithm o algoritmo hash seguro) que es un diseño de la *NSA* inspirado en MD4. SHA-1 proporciona un resumen de 160 bits y debería ser más segura que MD4 y MD5, debido a que sus resúmenes son más largos que los de las otras dos funciones; no obstante, ha sido rota recientemente por Google, demostrando la posibilidad de crear 2 mensajes distintos con el mismo resumen (véase [41]).

SHA-2

Como mejora a SHA-1, el NIST estandarizó SHA-2 en 2001. Es una evolución de este, también diseñado por la NSA, que tiene dos variantes distintas: una con un resumen de 256 bits (diseñada especialmente para procesadores de 32 bits) y otra con un resumen de 512 (diseñada para procesadores de 64 bits). Ambas versiones son consideradas seguras en la actualidad y gozan de gran popularidad (véase [42]).

2.3.4.3 Otras funciones hash

RIPEMD-160

El algoritmo RIPEMD-160 (Integrity Primitives Evaluation Message Digest, véase [43]) es un algoritmo hash diseñado para sustituir a MD4 y MD5. Genera un Hash de 20 bytes (160 bits) y, entre otros protocolos, se utiliza en la criptomoneda basada en *blockchain* Bitcoin (véase [44]).

Otros algoritmos que han gozado de cierta popularidad son: HAVAL [45], Snefru [46], GOST [47], MDC-2 [48], N-Hash [49], Panamá [50], Spectral [51], Tangle [52], CubeHash [53] o Fugue [54].

2.3.4.4 Códigos de autenticación

Los códigos de autenticación de mensaje o funciones MAC (Message Authentication Code), son funciones hash que además emplean una clave conocida tanto por el remitente como el destinatario. Se dividen en los siguientes grupos:

- CBC-MAC. Este algoritmo funciona cifrando el mensaje en modo CBC, y desechando todo el texto cifrado excepto el último bloque. Con esta construcción se tiene como objetivo convertir un algoritmo de cifrado simétrico de bloques en una función de autenticación de mensaje (véase [55]).
- HMAC. Consiste en utilizar una función hash para implementar una función MAC. La opción más popular es la de usar HMAC-SHA-256. Se basa en el uso de una función de hash más la aplicación de una clave secreta para generar un código de autenticación de mensaje (véase [56]).
- UMAC. Conocido como hash universal, es un método de construcción de MAC basado en la elección de forma aleatoria de una función hash de entre un conjunto disponible. Se especifica en el RFC 4418 (véase [57]).

2.3.5 Aplicaciones de las funciones resumen

Las funciones hash criptográficas tienen múltiples aplicaciones prácticas para solucionar problemas que se generan en el desarrollo y funcionamiento de sistemas seguros. A continuación, se detallan las más relevantes.

Integridad de la información

Dado que al variar el mensaje también varía el resumen, podemos utilizar las funciones hash para garantizar la integridad de los datos. Se puede aplicar a un mensaje o a varios para verificar que los mensajes enviados llegan sin modificación, duplicación, borrado o cualquier tipo de alteración.

Firmas digitales

El protocolo de firma digital permite al receptor determinar que el emisor es quien dice ser (autenticación y no repudio de origen) y verificar que el mensaje no ha sufrido modificaciones tras haber sido firmado (integridad).

Puesto que firmar el mensaje completo es muy lento (hace uso de criptografía de clave pública o asimétrica) se puede, en su lugar, firmar el resumen del mensaje que suele ser de tamaño más reducido y, por lo tanto, más rápido de firmar.

Gestión de Contraseñas

No resulta seguro guardar las contraseñas de los usuarios en claro en una base de datos que podría ser robada por cualquier atacante. Para evitar esto, se puede hacer uso de funciones hash con el objetivo de guardar el resumen de la contraseña en lugar de la contraseña en sí. De esta forma, si la base de datos con las contraseñas fuera robada, los atacantes sólo tendrían acceso a los resúmenes que carecerían de utilidad puesto que las funciones hash no son invertibles.

Derivación de claves

Si bien las funciones hash pueden parecer ser una opción idónea para la gestión de contraseñas, tiene el inconveniente de que las GPUs (tarjetas gráficas específicas) permiten calcular hashes a mucha velocidad, posibilitando a un atacante probar muchas contraseñas muy rápido e, incluso, probar todas las contraseñas posibles de una determinada longitud.

Para evitar el ataque por GPUs, han surgido diseños de funciones hash que tienen un coste computacional y espacial configurable, estableciendo cuanto tiempo tardan o qué cantidad de memoria utilizan. Este tipo de funciones se llaman PBKDF (Password Based Key-Derivation Function o función de derivación de clave basada en contraseña) y son, básicamente, funciones hash más lentas y que pretenden ralentizar a un atacante. También sirven para obtener una clave a partir de una contraseña.

Entre los algoritmos PBKDF más comunes, tenemos cuatro:

- *PBKDF2.* Esta función es un estándar de internet del año 2000 y se basa en repetir mediante un bucle una función hash convencional (véase [11]). El número de repeticiones, y por tanto el tiempo de cómputo, es configurable. Si bien es un estándar empleado por Microsoft o Apple entre otros, es de las peores PBKDF frente a GPUs, ya que utiliza internamente funciones hash convencionales que se pueden calcular muy eficientemente en dichos dispositivos.

- *BCRYPT.* Esta PBKDF de 1999 utiliza el cifrador en bloque *blowfish* como parte de su diseño (véase [11]). Además, emplea una modesta cantidad de memoria RAM lo que ralentiza a las GPUs al limitar su paralelismo.
- *SCRYPT.* Se basa en el cifrador en flujo *Salsa20* y hace uso de grandes cantidades de memoria RAM (véanse [58] y [59]); esto lo hace mucho más efectivo frente ataques de GPUs y ha gozado de gran popularidad tanto para la gestión de contraseñas como en el mundo de las criptomonedas alternativas a *bitcoin* (que utiliza SHA-2) como *litecoin* o *dogecoin* (véase [60]).
- *ARGON2.* Se basa en la función hash BLAKE2b y, tras ser elegida como ganadora en el concurso *Password Hashing Competition*, está gozando de mucha popularidad en el campo de las funciones PBKDF. Esta función se trata en más detalle en la página 43.

Otras aplicaciones

Las propiedades de las funciones hash permiten su uso en múltiples aplicaciones: detección de intrusos en la red, antivirus, detección de modificación de ficheros del sistema operativo, construcción de generadores pseudoaleatorios, etc.

3 Estado del arte

Como se ha indicado con anterioridad, las funciones hash asocian una cadena binaria de tamaño fijo a cadenas de tamaño arbitrario. Es frecuente que no se requiera el mensaje completo para una operación, sino simplemente un identificador del mismo.

Para que una función hash pueda ser empleada en criptografía, se construye en base a características de seguridad concretas, como que el coste computacional asociado a encontrar dos entradas distintas que produzcan el mismo resumen o una entrada que produzca un resumen determinado sea muy alto.

En este capítulo se detallan las funciones hash y de derivación de clave destacadas en base a varios concursos internacionales recientes como el SHA-3, el cual duró varios años siendo promovido por el Instituto Nacional de Estándares y Tecnología (NIST); o el concurso PHC, cuyo objetivo ha sido encontrar nuevos o mejores algoritmos de derivación de claves.

3.1 Análisis del concurso SHA-3

A pesar de que SHA-2 es considerado seguro, el NIST creó el concurso SHA-3 (de 2008 a 2012, véanse [61] y [62]) para tener un diseño completamente nuevo que no tuviera la herencia de MD4 ni del esquema Merkle-Damgård. De esta forma, si se encontrara un ataque satisfactorio para SHA-2, SHA-3 sería inmune al ser radicalmente diferente.

Al inicio de la competición en 2008, el NIST había recibido 64 propuestas de todo el mundo, incluyendo universidades y grandes empresas (BT, IBM, Microsoft, Qualcomm y Sony, por nombrar algunas). De estas 64 propuestas, 51 cumplieron con los requisitos y participaron en la primera ronda del concurso (véase [63]).

Durante las primeras semanas, los criptoanalistas atacaron despiadadamente las presentaciones. En julio de 2009, el NIST anunció 14 candidatos para la segunda vuelta. Después de pasar 15 meses analizando y evaluando el desempeño de estos candidatos, el NIST eligió cinco finalistas:

- *BLAKE.* Un hash basado en un esquema Merkle-Damgård mejorado, cuya función de compresión se basa en un cifrador en bloque, que a su vez se basa en la función central del cifrador en flujo ChaCha, incluyendo una cadena de sumas, XORs y rotaciones de palabras. BLAKE fue diseñado por un equipo de investigadores académicos con sede en Suiza y el Reino Unido.
- *Grøstl.* Otro hash que emplea un esquema Merkle-Damgård mejorado cuya función de compresión utiliza dos permutaciones (o cifrados de bloque de clave fija) basadas en la función central del cifrado de bloque AES. Grøstl fue diseñado por un equipo de siete investigadores académicos de Dinamarca y Austria.
- *JH.* Una construcción de función de esponja modificada en la que se inyectan bloques de mensajes antes y después de la permutación en lugar de justo antes. La permutación también realiza operaciones similares a un cifrador en bloque de sustitución y permutación. JH fue diseñado por un criptógrafo de una universidad de Singapur.
- *Keccak.* Una función de la esponja cuya permutación realiza solamente operaciones binarias basadas en bits. Keccak fue diseñado por un equipo de cuatro criptógrafos que trabajan para una empresa de semiconductores con sede en Bélgica e Italia, e incluyó a uno de los dos diseñadores de AES.

- *Skein.* Una función hash basada en un modo de funcionamiento diferente al de Merkle-Damgård, y cuya función de compresión se basa en un nuevo cifrado por bloques que utiliza únicamente la suma de números enteros, XOR y la rotación de palabras. Skein fue diseñado por un equipo de ocho criptógrafos de la academia y la industria, todos menos uno de los cuales tienen su sede en los Estados Unidos, incluyendo al renombrado Bruce Schneier.

Después de un extenso análisis de los cinco finalistas, el NIST anunció un ganador: Keccak. El informe del NIST recompensó a Keccak por su "diseño elegante, gran margen de seguridad, buen rendimiento general, excelente eficiencia en hardware y su flexibilidad".

El NIST no planea retirar SHA-2 o eliminarlo de los estándares actuales de hash. El propósito de SHA-3 es que sea intercambiable directamente por SHA-2 en aplicaciones actuales si es necesario; mejorando, de esta forma, significativamente la robustez del portfolio de herramientas de hash que ofrece el NIST. Se analizan, a continuación, tanto Keccak como el resto de los finalistas.

3.1.1 Keccak

El algoritmo Keccak es obra de Guido Bertoni, Joan Daemen (quien también diseñó el cifrador en bloque Rijndael junto con Vincent Rijmen [17]), Michael Peeters y Gilles Van Assche. Se basa en los diseños anteriores de funciones hash PANAMÁ y RadioGatún (véase [64]).

La estructura de SHA-3 es muy diferente de la de SHA-1 y SHA-2. La idea clave detrás de SHA-3 se basa en permutaciones sin clave, a diferencia de otras construcciones típicas de funciones de hash que utilizaban permutaciones con clave. Keccak tampoco hace uso de la transformación Merkle-Damgård que se utiliza comúnmente para aceptar mensajes de entrada de longitud arbitraria en funciones hash. En Keccak se utiliza un nuevo enfoque llamado construcción de esponja y compresión (véase [65]) que es un modelo de permutación aleatoria. Se han estandarizado diferentes variantes de SHA-3, como SHA-3-224, SHA-3-256, SHA-3-384, SHA-3-512, SHAKE-128 y SHAKE-256. Las funciones SHAKE-128 y SHAKE-256 son funciones de salida extensibles (o XOF, véase [66]) que permiten que la salida se extienda a cualquier longitud deseada.

Respecto al rendimiento, SHA3-256 en x86-64 es capaz de funcionar a 12 ciclos por byte dependiendo de la CPU (véase [67]). El algoritmo SHA-3 ha sido criticado por ser lento en las arquitecturas que no tienen instrucciones específicas para ejecutar las funciones de Keccak más rápido; cabe destacar que SHA2-512 es más del doble de rápido que SHA3-512 y que SHA-1 es más del triple de rápido en un procesador Intel Skylake. Los autores de Keccak han reaccionado a esta crítica sugiriendo el uso de SHAKE128 y SHAKE256 en lugar de SHA3-256 y SHA3-512, obteniendo un rendimiento a la par con SHA2-256 y SHA2-512 pero reduciendo la resistencia de preimagen a la mitad y manteniendo el nivel de resistencia a colisiones. No obstante, en las implementaciones de hardware, SHA-3 es notablemente más rápido que todos los demás finalistas y también más rápido que SHA-2 o SHA-1 (véase [68]).

En 2016, el mismo equipo que diseñó Keccak introdujo versiones más rápidas con un número reducido de rondas (12 y 14 rondas, en lugar de las 24 de SHA-3). Estas alternativas, KangarooTwelve y MarsupilamiFourteen (véase [69]), pueden explotar la ejecución en paralelo gracias a la utilización de hash en árbol. Estas funciones difieren de ParallelHash (véase [70]) en lo que respecta al paralelismo y son más rápidas para mensajes cortos.

Estos algoritmos de mayor velocidad no son parte de SHA-3 (ya que son un desarrollo posterior), y por lo tanto no son compatibles con los estándares, pero son tan seguros como las funciones de SHA-3, porque utilizan la misma permutación de Keccak y no hay ataques a Keccak de 12 rondas.

3.1.2 BLAKE

BLAKE y BLAKE2 son funciones de hash criptográficas basadas en el cifrador en flujo ChaCha de Dan Bernstein (véase [71]), pero con la variación de que se añade antes de cada ronda una copia permutada del bloque de entrada, combinada mediante or exclusiva con algunas constantes de ronda. Al igual que el SHA-2, hay dos variantes que difieren en el tamaño de las palabras: BLAKE-256 y BLAKE-224 usan palabras de 32 bits y producen tamaños de resumen de 256 bits y 224 bits, respectivamente; mientras que BLAKE-512 y BLAKE-384 usan palabras de 64 bits y producen tamaños de resumen de 512 bits y 384 bits, respectivamente. BLAKE combina repetidamente un valor de hash de 8 palabras con 16 palabras de mensaje, truncando el resultado de ChaCha para obtener el siguiente valor de hash (véase [72]).

La seguridad puede ser lo más importante, pero la velocidad es lo segundo. En muchos casos, un desarrollador no cambiaría de MD5 a SHA-1 simplemente porque MD5 es más rápido, o de SHA-1 a SHA-2 porque SHA-2 es notablemente más lento que SHA-1.

Desafortunadamente, SHA-3 no es más rápido que SHA-2, y debido a que el SHA-2 sigue siendo seguro, hay pocos incentivos para actualizar al SHA-3. BLAKE2 (véase [73]), lanzada tras la competición SHA-3, ofrece un rendimiento superior a SHA-1 y SHA-2 pero con mucha mayor seguridad. BLAKE2 fue diseñado con los siguientes objetivos:

- Al menos tan seguro como SHA-3.
- Más rápido que todos los estándares hash anteriores, incluyendo MD5.
- Adecuado para su uso en aplicaciones modernas, y capaz de procesar grandes cantidades de datos, ya sea pocos mensajes grandes o muchos pequeños
- Debe ser adecuado para su uso en procesadores modernos que soportan computación paralela en sistemas multinúcleo, así como paralelismo a nivel de instrucción dentro de un único núcleo.

Tras el proceso de diseño, se obtuvieron como resultado un par de funciones hash: BLAKE2b, optimizada para plataformas de 64 bits y resúmenes de 1 a 64 bytes; y BLAKE2s, optimizada para plataformas de 8 a 32 bits y resúmenes de 1 a 32 bytes.

3.1.3 Grøstl

Grøstl es una función de hash criptográfica (véase [74]) diseñada por Praveen Gauravaram, Lars Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer y Søren S. Thomsen. Utiliza la misma caja de sustitución que AES en una construcción diferente. Los autores especificaban un rendimiento de hasta 21 ciclos por byte en un Intel Core 2 Duo.

Como otras funciones hash de la familia MD5 o SHA, Grøstl divide la entrada en bloques y opera iterativamente sobre ellos. Sin embargo, mantiene un estado de hash de al menos el doble del tamaño de la salida final (512 o 1024 bits), que sólo se trunca al final del cálculo del resumen.

La función de compresión se basa en un par de permutaciones de 256 o 512 bits que están basadas en AES, aunque operando en matrices de 8x8 u 8x16 bytes, en lugar de 4x4. Al igual

que en AES, cada ronda consiste en las cuatro operaciones *AddRoundKey*, *SubBytes*, *ShiftBytes* y *MixColumns.* Se recomiendan entre 10 y 14 rondas en función del tamaño del resumen.

3.1.4 JH

JH es una función resumen criptográfica diseñada por Hongjun Wu (véase [75]). Tiene un estado de 1024 bits y trabaja sobre bloques de entrada de 512 bits. Cada bloque de entrada *i* se procesa siguiendo los pasos siguientes:

- Realizar el XOR de *i* con la mitad izquierda del estado.
- Aplicar una permutación sin clave durante 42 rondas al estado. Esto incluye la división del estado en 256 bloques de 4 bits que se procesan con dos cajas de sustitución 4x4 y un código separable de distancia máxima (MDS) sobre GF(2^4).
- Realizar el XOR de *i* con la mitad derecha del estado.

El resumen se obtiene mediante truncamiento del estado final de 1024 bits. Se puede implementar adecuadamente con el conjunto de instrucciones SSE2 de Intel (véase [76]), obteniendo un rendimiento de 16.8 ciclos por byte.

3.1.5 Skein

Skein (véase [77]) fue creado por Bruce Schneier, Niels Ferguson, Stefan Lucks, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas y Jesse Walker.

Skein se basa en el cifrador en bloque Threefish (véase [78]), realizando la compresión mediante el modo de encadenamiento de iteración de bloques único (UBI), una variante del modo hash de Matyas-Meyer-Oseas (véase [79]).

Skein soporta tamaños de estado interno de 256, 512 y 1024 bits, y tamaños de salida arbitrarios. Los autores indican un rendimiento de 6 ciclos por byte para cualquier tamaño de salida en un Intel Core 2 Duo en modo de 64 bits.

El núcleo de Threefish se basa en una función MIX que transforma 2 palabras de 64 bits usando una sola suma, rotación por una constante y XOR. El modo de encadenamiento UBI combina un valor de encadenamiento de entrada con una cadena de entrada de longitud arbitraria y produce una salida de tamaño fijo.

La no linealidad de Threefish proviene enteramente de la combinación de operaciones de suma y XOR; no utiliza cajas de sustitución. La función está optimizada para procesadores de 64 bits y los autores definen características opcionales como la posibilidad de funcionar a modo de hash aleatorio, árbol de hash paralelizable, cifrado en flujo, personalización y una función de derivación de clave.

En octubre de 2010, se publicó un ataque (véase [80]) que combina el criptoanálisis rotacional con el ataque de rebote. El ataque encuentra colisiones rotacionales para 53 de 72 rondas en Threefish-256, y 57 de 72 rondas en Threefish-512. Ataque que se puede extender a Skein. Para evitar cualquier duda sobre la seguridad de Skein, sus autores ajustaron la constante asociada a las claves de ronda para hacer este ataque menos efectivo.

3.2 Análisis del concurso PHC

La competición *Password Hashing Competition* (PHC, véase [81]) se realizó entre 2013 y 2015 centrándose en algoritmos de gestión de contraseñas o de derivación de clave (PBKDF) con un proceso muy similar al de los concursos AES y SHA-3 del NIST.

La principal motivación para el desarrollo de esta competición fue la escasa disponibilidad de soluciones criptográficas sencillas y seguras para el almacenamiento de contraseñas adecuado en la autentificación de usuarios de servicios online.

En el primer trimestre de 2013 se hizo pública la convocatoria con límite hasta 31 de marzo de 2014, durante este periodo se presentaron veinticuatro candidatos; en diciembre de 2014 se hizo una selección a nueve finalistas con la publicación de un breve informe y en julio de 2015 Argon2 fue anunciado como ganador, dando un reconocimiento especial a los cuatro finalistas Catena, Lyra2, Makwa, yescrypt. Estos algoritmos se analizan a continuación.

3.2.1 Argon2

Argon2 (véase [82]), desarrollado en la Universidad de Luxemburgo por Alex Biryukov, Daniel Dinu, y Dmitry Khovratovich, es una nueva función de derivación de clave y, tras ser anunciado como el ganador de PHC, es el máximo exponente actual en el campo de la autenticación de usuarios mediante contraseñas y de funciones que permiten ajustar el equilibrio entre los costes computacionales y espaciales (véase [83]).

Argon2 se basa internamente en la función hash BLAKE2b y proporciona tres variantes, la principal *argon2id* y dos subvariantes *argon2i* y *argon2d*:

- *Argon2d.* Utiliza acceso a memoria dependiente de los datos (contraseña) y es especialmente adecuado para criptomonedas como algoritmo de prueba de trabajo y en aquellas aplicaciones que no sean susceptibles de ataques por temporización.
- *Argon2i.* Es la versión resistente a ataques de temporización de Argon2. Utiliza el acceso a la memoria independiente de los datos, que es el preferido para el hash de contraseñas y la derivación de claves basada en contraseñas. Argon2i requiere más pasadas sobre la memoria para protegerse de los ataques que utilicen hardware específico.
- *Argon2id.* Es una versión híbrida de Argon2 que combina Argon2i y Argon2d. Utiliza el acceso a la memoria independiente de los datos para la primera mitad de la primera iteración sobre la memoria y el acceso a la memoria dependiente de los datos para el resto. Argon2id es resistente a los canales laterales (ataques de temporización) y proporciona una mejor resistencia a ataques que exploten el equilibrio memoria/tiempo que Argon2i.

Si bien no existen ataques conocidos para Argon2d, se han publicado ataques para Argon2i que obligan a realizar un mínimo de 10 pasadas con este algoritmo (véase [84]).

3.2.2 CATENA

CATENA, desarrollado por Sascha Schmidt, es un marco de gestión de contraseñas nuevo y comprobadamente seguro, que cuenta con propiedades de vanguardia; su versión actual es catena-V3.3 que se lanzó el 2 de noviembre de 2015 (véase [85]).

CATENA soporta contraseñas y sales de longitud arbitraria, proporciona parámetros para ajustar el uso de memoria y tiempo (*pepper* permite mantener el secreto de parte de la sal, *garlic* controla el coste espacial y *lambda* especifica el número de pilas de la estructura interna) y produce salidas de 32 bytes.

Además, el algoritmo está diseñado para cumplir las siguientes propiedades de seguridad:

- Características criptográficas estándar como resistencia a ataques de preimagen, resistencia a colisiones, inmunidad a la extensión de longitud y salida indistinguible de una secuencia aleatoria.
- Resistencia a ataques masivos en paralelo utilizando hardware específico con cantidades limitadas de memoria rápida como tarjetas gráficas, chips a medida y programables.
- Resistencia a ataques de canal lateral, como los de temporización de caché.

3.2.3 Lyra2

Lyra2 fue desarrollada por Marcos Simplicio, Leonardo Almeida, Ewerton Andrade, Paulo dos Santos y Paulo Barreto, en la Universidad de Sao Paulo (véase [12]).

Lyra2 es una mejora de Lyra, propuesta anteriormente por los mismos autores. Lyra2 preserva la seguridad, eficiencia y flexibilidad de su predecesor, incluyendo la capacidad de configurar la cantidad deseada de memoria, el tiempo de procesamiento y el paralelismo que utilizará el algoritmo y la capacidad de proporcionar un alto uso de memoria con un tiempo de procesamiento similar a otras PBKDF existentes como, por ejemplo, Scrypt. Además, aporta importantes mejoras en comparación con su predecesor, entre las que se pueden destacar:

- Un nivel de seguridad superior frente a ataques que exploten el equilibrio memoria/tiempo.
- Permite que los usuarios legítimos hagan un mejor uso de las capacidades de paralelismo disponibles en las plataformas de computación actuales.
- Incluye modificaciones que aumentan el coste de los ataques con hardware dedicado.
- Dificulta los ataques de canal lateral que hacen uso de dispositivos asequibles de almacenamiento masivo.

Internamente, la memoria del esquema se organiza como una matriz que se espera que permanezca en memoria durante todo el proceso de hashing de contraseñas. La construcción y utilización de la matriz se realiza mediante la combinación de las operaciones de absorción,

exprimido y dúplex de una función de esponja subyacente. Esta función esponja puede elegirse en función de la situación; por ejemplo, BLAKE2b para software, Keccak para hardware, etc.

3.2.4 Makwa

Makwa, desarrollada por Thomas Pornin, es una función de hash dedicada al procesamiento de contraseñas en fichas con fines de verificación de contraseñas y de derivación de claves simétricas apropiadas para algoritmos criptográficos simétricos. Además, ofrece una función adicional, que generalmente no se proporciona por parte de las clásicas funciones hash donde el hash puede ser delegado a un sistema externo no necesariamente de confianza. La delegación puede ofrecer al defensor un enorme aumento en la potencia de cálculo disponible para el hashing de contraseñas, permitiendo así una productividad mucho más alta (véase [10]).

El núcleo principal de Makwa es una secuencia de cuadrados modulares. El coste computacional es proporcional al número de cuadrados sucesivos. El incremento del coste computacional es tan simple como incrementar el número de cuadrados a calcular. Esta estructura algebraica permite la delegación, de una manera similar a las firmas ciegas.

Entre las características principales de Makwa, se pueden destacar:

- Utiliza aritmética de grandes números con precisión arbitraria (para el cálculo de cuadrados modulares).
- Usa HMAC-DRBG (un estándar bien conocido del NIST, que se basa en HMAC sobre una función hash, véase [86]).
- Admite la delegación del procesamiento a sistemas no necesariamente de confianza, lo que permite optimizar el rendimiento y mejorar la resistencia a ataques masivamente paralelos.

3.2.5 Yescrypt

Yescrypt, desarrollado por Alexander Peslyak, se basa en Scrypt, incorporando una serie de mejoras y modificaciones con el objetivo de optimizar el uso del ancho de banda de memoria (véase [13]).

La característica más destacable de Yescrypt para despliegues a gran escala es la inicialización opcional y la reutilización de una tabla de búsqueda grande, que normalmente ocupa al menos decenas de gigabytes de RAM y forma esencialmente una ROM específica para cada servidor o servicio. Esto limita los ataques basados en hardware específico, o masivamente paralelo como los nodos de botnets (véase [87]).

Entre las ventajas de Yescrypt frente a Scrypt o Argon2, se encuentran: una mejor resistencia frente a ataques *offline*, características opcionales disponibles (cifrado de hashes, actualización de la seguridad independientemente del cliente, etc.) y seguridad criptográfica basadas en primitivas estandarizadas por el NIST.

Entre las desventajas, destacan la mayor complejidad, que es susceptible a ataques por temporización de caché y que su disponibilidad en librerías para el desarrollo en diferentes lenguajes de programación es escasa.

4 Diseño y desarrollo

Las funciones de gestión de contraseñas, o PBKDF, son funciones hash criptográficas especializadas que presentan parámetros de coste ajustables con el objetivo de frustrar los ataques por fuerza bruta sobre bases de datos con resúmenes de contraseñas, incluso cuando se emplea hardware especializado o unidades de procesamiento gráfico de propósito general.

En este capítulo se propone el diseño de una función resumen de gestión de contraseñas basada en el estándar de cifrado en bloque AES utilizando el modo contador (CTR), analizando las características de rendimiento, seguridad y complejidad y obteniendo resultados prometedores en comparativa con el estado del arte en términos de PBKDF: Scrypt y Argon2.

4.1 Motivación

Cabe destacar que la investigación realizada está fundamentada en la necesidad y en la demanda constante de mantener altos niveles de seguridad de los sistemas criptográficos, específicamente en el servicio de la gestión de contraseñas usado en la mayoría de los sistemas de información y servicios de Internet, en combinación con otras técnicas disponibles actualmente como la biometría.

Existe abundante literatura respecto a estas funciones de gestión de contraseñas, o funciones de derivación de claves basadas en contraseña (PBKDF), con muchas publicaciones recientes (véanse [7], [9], [10], [13], [88], [89], [90], [91]) que mejoran el conocido estándar PBKDF2 (véase [14]).

Además de en el hash de contraseñas y la obtención de claves, las PBKDF han encontrado un vasto número de aplicaciones en el campo de las criptomonedas y los algoritmos de cadena de bloques o *blockchain* como funciones de prueba de trabajo en muchos de estos diseños (véase [15]). La aparición de la criptomoneda *bitcoin* (véase [92]), al ser de libre acceso y proporcionar seguridad criptográfica, ha logrado despertar el interés tanto del público como de empresas para empezar a desarrollar sus propios productos y aplicaciones. Con el objetivo de mantener la democratización de la gestión de las criptomonedas, han surgido variantes de bitcoin que utilizan PBKDF como función de prueba de trabajo en el protocolo blockchain, limitando la ventaja de las grandes corporaciones con acceso a hardware específico (véase [60]).

Cabe destacar que los finalistas de la competición de hash de contraseñas (Argon2 [91], Catena [9], Makwa [10], Lyra2 [90] y yescrypt [13]) son funciones altamente relevantes e íntimamente relacionadas con la función PBKDF propuesta en este trabajo.

Frecuentemente, la autenticación de usuarios mediante contraseñas se procesa mediante una función hash criptográfica sencilla junto a una cadena aleatoria llamada sal. En la actualidad, los atacantes han tenido éxito utilizando tarjetas gráficas (GP-GPUs) e incluso utilizando hardware especializado, hallando contraseñas mediante fuerza bruta. Es por esto que resulta necesario emplear funciones hash criptográficas que soporten parámetros de espacio de memoria y coste temporal ajustables, como las PBKDF, para contrarrestar estos ataques (véase [6], [11]).

4.2 Diseño

A continuación, se describen los parámetros y elementos de las funciones; así como el diseño de la función PBKDF original y sus dos siguientes versiones de funciones propuestas y optimizadas. Exponiendo así tres variantes: la versión original AESCTR-o (publicada en [88]), la etapa de optimización intermedia AESCTR-i y la optimización final AESCTR-f (ambas publicadas en [93]). Para mayor claridad, esta última (AESCTR-f) es descrita en el capítulo siguiente.

La notación de pseudocódigo está basada en el lenguaje Go ([véase [94]), ya que resulta especialmente adecuado para la definición de las etapas de inicialización y de salida de la propuesta.

Las funciones PBKDF propuestas están basadas en el uso del estándar AES en modo contador (CTR), actuando como un generador de números pseudoaleatorios (PRNG). La PBKDF se compara favorablemente en rendimiento y seguridad con estándares comprobados, como Scrypt [7] o Argon2 [91], dado que se explota la ventaja que tiene AES de ser acelerado por hardware en la mayoría de los procesadores modernos, permitiendo así protegerse frente ataques basados en GPU o en hardware específico, equilibrando el coste computacional entre defensor y atacante.

La función de gestión de contraseñas propuesta puede ser especialmente útil en aplicaciones donde se requiera una autenticación segura y sea valioso disponer de un rendimiento configurable, como en escenarios que involucren tecnología entre pares, aplicaciones móviles, computación embebida, redes de sensores remotos o Internet de las cosas, entre otros.

4.2.1 Versión inicial: AESCTR-o

El estándar de cifrado en bloque AES es más lento de forma nativa que algunos cifradores en flujo actuales, pero tiene la ventaja de ser acelerado por hardware en muchos procesadores

modernos, lo que hace que sea extremadamente rápido, con velocidades superiores a 1 GiB/s en sistemas actuales.

Es una excelente primitiva básica sobre la que construir muchas herramientas criptográficas, incorporando en su diseño la velocidad y seguridad que aporta este estándar.

En esta sección se detalla el diseño de la función inicial, con los parámetros de entrada de la función denominada *AESCTR-o* de gestión de contraseñas, así como los elementos esenciales que forman el núcleo de la función e incluyendo el pseudocódigo detallado para las etapas de inicialización y salida del algoritmo.

4.2.1.1 Elementos

Los elementos que conforman la función inicial se muestran en la Tabla 1, donde en la función fC() se emplea el estándar AES-128 en modo CTR; siendo la primitiva criptográfica básica en el diseño de la función propuesta y utilizándose principalmente como un PRNG.

Tabla 1. Elementos de AESCTR-o

M[]:	Matriz de bytes de longitud pmen por plen bytes. Constituye el principal búfer de memoria en el algoritmo.
out[]:	Matriz de plen bytes, contiene el hash de salida.
fH():	Función hash criptográfica segura con un tamaño de resumen de 256 bits utilizado para la semilla. En este caso, se emplea el estándar SHA3-256.
fC():	Cifrador por bloque AES-128 estándar, en modo CTR que actúa como un PRNG.

4.2.1.2 Parametrización

Los parámetros que conforman la función se detallan a continuación:

Tabla 2. Parámetros para AESCTR-0

pass[]: Matriz de bytes de longitud arbitraria, contiene la contraseña.

salt[]: Matriz de bytes de longitud arbitraria, contiene la sal aleatoria. Se recomienda un mínimo de 32 bytes.

plen: Entero positivo que indica la longitud en bytes del resumen de salida. Se recomienda un mínimo de 32 bytes.

pmem: Entero positivo que indica el número de entradas (de plen bytes) que contiene la matriz principal (elemento M[]). Este parámetro modula el coste espacial (memoria). Su valor mínimo posible es 1.

ptime: Entero positivo que indica el número de pasadas sobre la matriz principal. Este parámetro modula el coste temporal. Ha de ser 1 como mínimo.

4.2.1.3 Inicialización

Se define a continuación la inicialización para la función inicial (AESCTR-0), que devuelve una matriz de bytes, con el hash de salida. Se utiliza, además, la llamada a la función len(), que devuelve la dimensión de vectores, evitando tener que instanciar la dimensión por separado.

```
func Key(pass, salt []byte, plen, pmem, ptime int)
```

En la Tabla 3, se muestra la codificación de la fase de inicialización, se resume la contraseña junto con la sal, separada por un byte cero; esto está destinado a evitar colisiones donde la contraseña y la sal se concatenan directamente dando como resultado los mismos datos de entrada. Luego, los primeros 16 bytes de la semilla se usan como clave de 128 bits para AES y los segundos 16 bytes de la semilla conforman el vector de inicialización de 128 bits para el modo CTR. Así, los buffers out[] y M[] son cifrados usando la instancia inicializada de AES. Por defecto, asumimos que este búfer se inicializa a cero mediante la función *make()* antes del cifrado.

Tabla 3. Pseudocódigo para la inicialización de AESCTR-o

```
// semilla de 256 bit a partir de contraseña y sal
fH := sha3.New256()
fH.Write(pass)
fH.Write(0)
fH.Write(salt)
seed := fH.Sum(nil)

// inicialización de AES-128-CTR a partir de la semilla
blk := aes.NewCipher(seed[0:16])
fC := cipher.NewCTR(blk, seed[16:32])

//se inicializa la salida cifrándola (pseudoaleatorio)
out := make([]byte, plen)
fC.XORKeyStream(out, out)

//se inicializa la matriz cifrándola (pseudoaleatorio)
M := make([]byte, pmem*plen)
fC.XORKeyStream(M, M)
```


4.2.1.4 Salida

El algoritmo de salida es relativamente sencillo, como se puede comprobar en el pseudocódigo de la Tabla 4 y el diagrama de flujo de la Figura 5. Para cada entrada en M[], se obtiene un índice pseudoaleatorio a partir de los primeros 8 bytes (64 bits) de out[] que, a su vez, se emplea para elegir una entrada de M[] que se combine con el estado actual de out[]. Esta operación es una sustracción a nivel de bytes (Z_{256}) ya que no es conmutativa. A continuación, el resultado de esta operación se cifra y se almacena en out[].

El bucle exterior es el número de veces (o pasadas a través de M[]) que se realizan en este proceso. La salida definitiva de la función es el último estado de out[] y, puesto que se trata de la salida directa de un cifrado AES, toda la función es tan segura como AES.

Tabla 4. Pseudocódigo para la salida de AESCTR-o

```
for t := 0; t < ptime; t++ {
        for m := 0; m < pmem; m++{
        i := (int(out[0:8])%pmem)*plen
                    for o := 0; o < len (out), o++ {
                        M[i+o] -= out[o]
                    }
        fC.XORKeyStream(out, M[i:i+plen])
        }
}
```

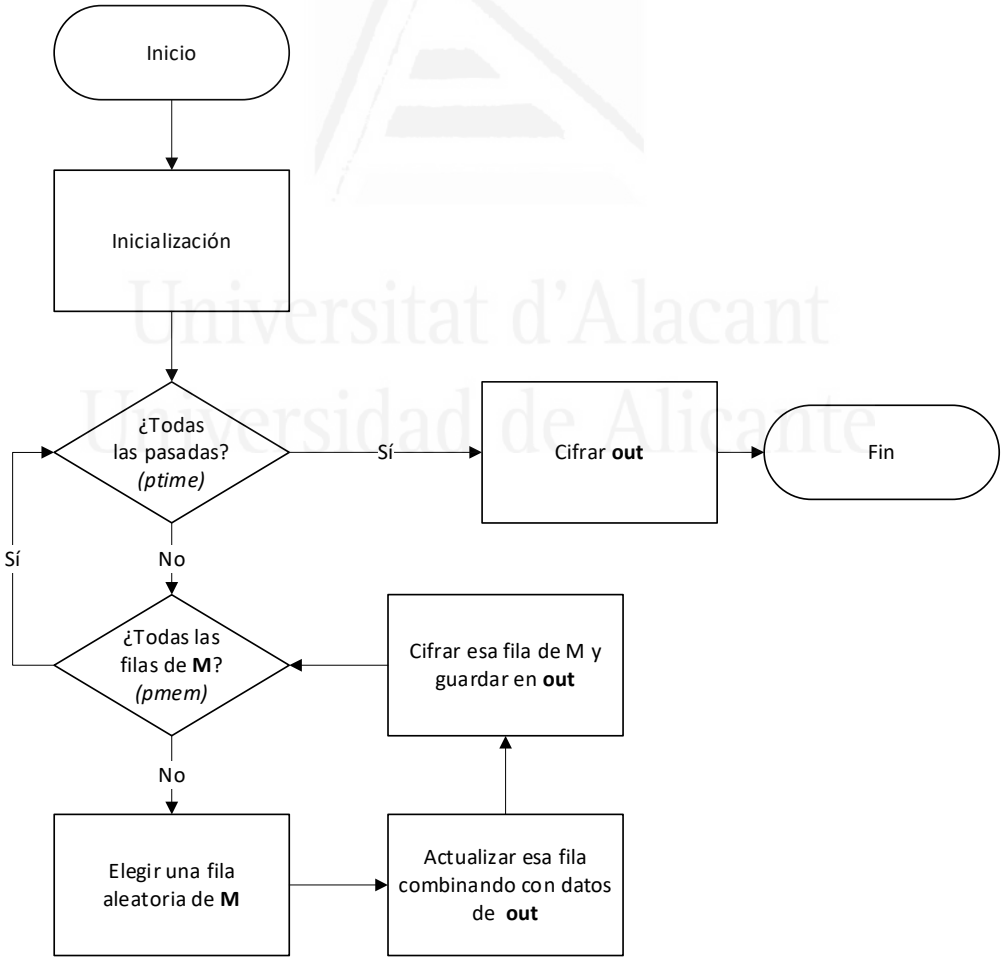


Figura 5. Diagrama de flujo para la salida de AESCTR-o

4.2.1.5 Resultados

A continuación, se detallan los aspectos de rendimiento con respecto a la complejidad espacial y temporal, así como una comparación con la reconocida función PBKDF Scrypt.

Estas pruebas fueron implementadas en el lenguaje de programación Go (versión 1.8.1) y ejecutadas en un computador con procesador Intel Core i5 de 2.6 GHz (que soporta AES acelerado por hardware) y con 16 GiB de RAM; las contraseñas y cadenas de sal tienen una longitud de 32 bytes (256 bit), así como la salida que es de 32 bytes también. Cada prueba se realizó 100 veces.

Complejidad espacial

En la figura Figura 6, se puede ver el coste computacional de la función analizada cuando el parámetro *pmem* aumenta exponencialmente de 2^8 a 2^{23} entradas de 32 bytes (o 8KiB a 256MiB de uso de memoria) con una sola pasada (ptime = 1). Los valores se pueden consultar en los anexos, página 95.

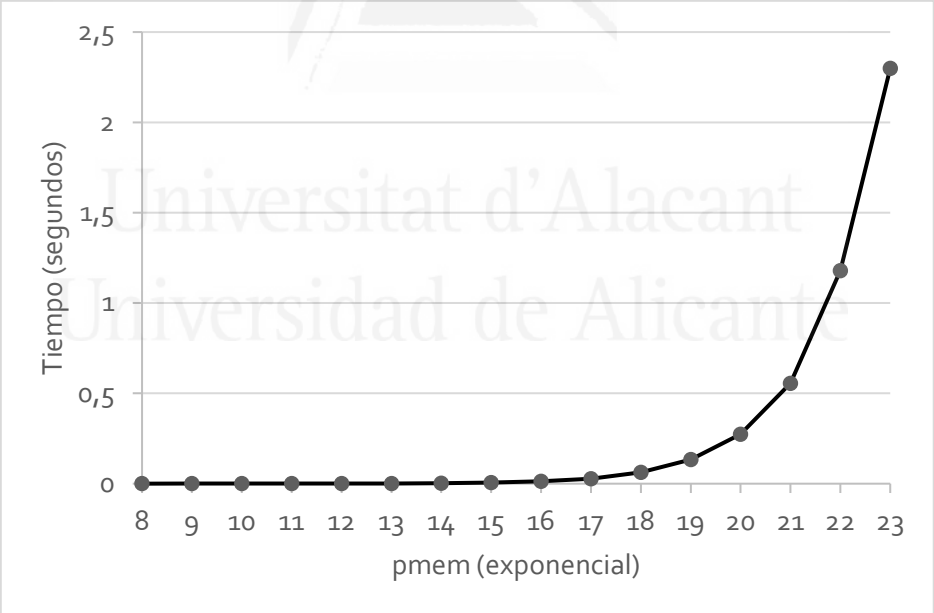


Figura 6. Coste computacional al incrementar la complejidad espacial

Se puede observar que se obtienen valores muy razonables de alrededor de 0.5 segundos o menos con *pmem* con valor 2^{21}. También hay un margen disponible significativo, en caso de que se deseara una mayor complejidad computacional.

Complejidad temporal

El tiempo de computo del parámetro ptime se muestra en la Figura 7. En este caso, ptime aumenta exponencialmente de 1 a 2^{15} pasadas y pmem es fijado en 256 entradas de 32 bytes, que equivalen a 8KiB de memoria. Se tiene un rendimiento razonable con tiempos de alrededor de 0.5 segundos o menos con hasta 2^{14} pasadas. Los valores se pueden consultar en los anexos, página 96.

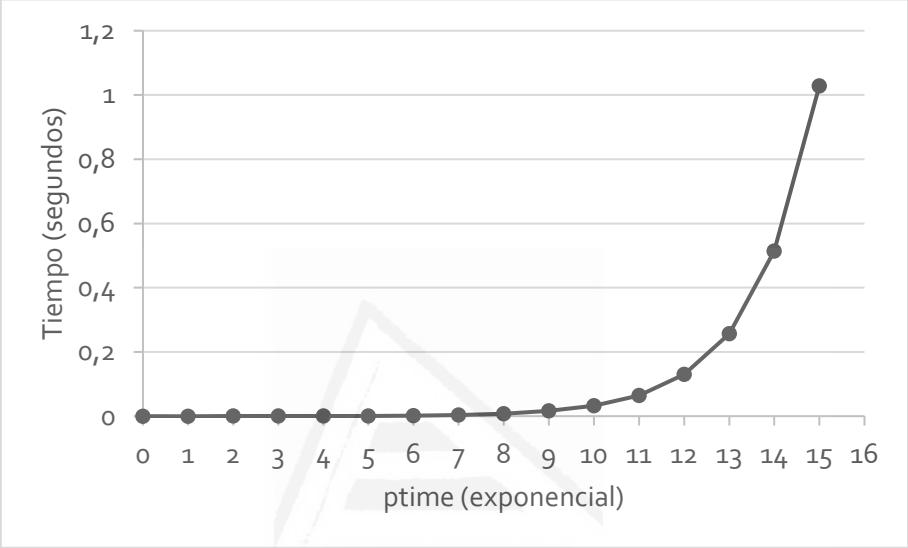


Figura 7. Coste computacional al incrementar la complejidad temporal

Complejidad combinada

En la Figura 8 se muestra el tiempo de ejecución asociado a la modulación de ambos parámetros, *pmem* y *ptime*, simultáneamente en un doble bucle. El número de entradas en la matriz *M[]* para el bucle interno va de 2^8 a 2^{15} y el número de pasadas durante el bucle externo va de 1 a 2^7. La serie esperada de aumento exponencial de la curva se observa claramente, mostrando un buen rango de rendimiento en la modulación de ambos parámetros de complejidad. El rendimiento también es muy razonable dentro de estos parámetros, con todos los tiempos alrededor de 0.5 segundos o menos. La cantidad máxima de memoria procesada es de 128 MiB con 2^{15} entradas de 32 bytes y 2^7 pasadas. Los valores se pueden consultar en los anexos, página 97.

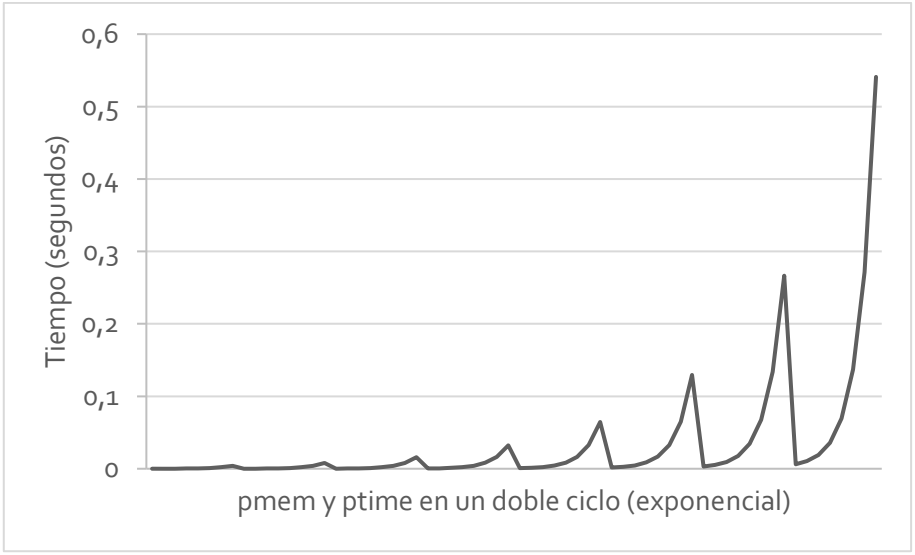


Figura 8. Coste computacional al incrementar ambas complejidades simultáneamente

4.2.1.6 Comparativa con Scrypt

Una de las funciones de gestión de contraseñas más utilizadas en los servicios actuales es Scrypt (véase [7]), ya que ha sido diseñada para resistir los ataques de GPU y además utiliza una cantidad configurable de memoria.

Se puede observar en la Figura 9, así como en los valores desplegados en la Tabla 5, que AESCTR-0 es significativamente más rápido que Scrypt cuando se procesan cantidades equivalentes de memoria. En este caso, se han empleado las implementaciones de la librería estándar de Go para AES que aprovecha la ventaja de la aceleración por hardware, y para Scrypt, que no emplea aceleración por hardware ya que usa la versión de 8 rondas de Salsa20 como primitiva criptográfica elemental.

Este análisis muestra que AES es una buena opción en esta aplicación en la mayoría de las máquinas modernas; por el contrario, en plataformas donde AES no está acelerado por hardware, el rendimiento puede ser considerablemente menos competitivo.

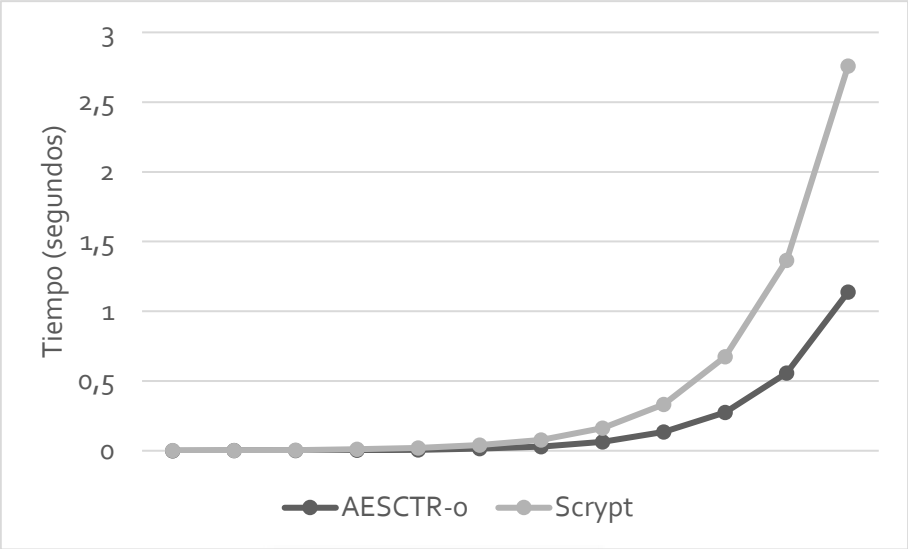


Figura 9. Comparativa con Scrypt para memoria equivalente

Tabla 5. Valores de la comparativa con Scrypt

pmem	AESCTR-0 (s)	Scrypt (s)
8	0,000391	0,000774
9	0,000780	0,001513
10	0,001562	0,003001
11	0,003431	0,006577
12	0,007240	0,013007
13	0,016000	0,024674
14	0,028405	0,049614
15	0,062817	0,098736
16	0,134064	0,198680
17	0,274017	0,400811
18	0,558009	0,806953
19	1,137899	1,620456

4.2.2 Optimización intermedia: AESCTR-i

Los parámetros de esta optimización intermedia coinciden en su totalidad con los de la versión inicial (véase la Tabla 2, página 53). Las variables del algoritmo tampoco cambian en esta optimización (véase la Tabla 1, página 52).

4.2.2.1 Inicialización

La inicialización de la función final es similar a la versión original AESCTR-o, como se muestra en la Tabla 6.

Tabla 6. Pseudocódigo para la inicialización de AESCTR-i

```
// semilla de 256 bit a partir de contraseña y sal
fH := sha3.New256()
fH.Write(pass)
fH.Write(0)
fH.Write(salt)
seed := fH.Sum(nil)

// inicialización de AES-128-CTR a partir de la semilla
blk := aes.NewCipher(seed[0:16])
fC := cipher.NewCTR(blk, seed[16:32])

//se inicializa la salida cifrándola (pseudoaleatorio)
out := make([]byte, plen)
fC. XORKeyStream(out, out)

//se inicializa la matriz cifrándola (pseudoaleatorio)
M := make([]byte, pmem*plen)
fC. XORKeyStream(M, M)
```

4.2.2.2 Salida

Para la salida de la optimización intermedia se modifica el bucle interno, intentando mejorar el rendimiento evitando los pasos de cifrado AES dentro de los bucles y simplemente cifrando el búfer out[] como último paso. Por esta razón, hay un segundo bucle interno que genera un nuevo índice para una fila diferente de M[] que se combina con el estado de out[]. En la Tabla 7, se han destacado las partes que difieren de la variante original (AESCTR-o) y se incluye el diagrama de flujo para esta versión en la Figura 10.

Tabla 7. Pseudocódigo para la salida de AESCTR-i

```
for t:=0; t<ptime; t++ {
    for m:=0; m<pmem; m++{
        i := (int(out[0:8])%pmem)*plen
        for o:=0; o<len(out), o++ {
            M[i+o] -= out[o]
        }
        i=(i*i)%pmen
        for o:=0; o<len(out);o++ {
            out[o]:-=(M[i+o]^out[o])
        }
    }
}
fC.XORKeyStream(out, M[i:i+plen])
```

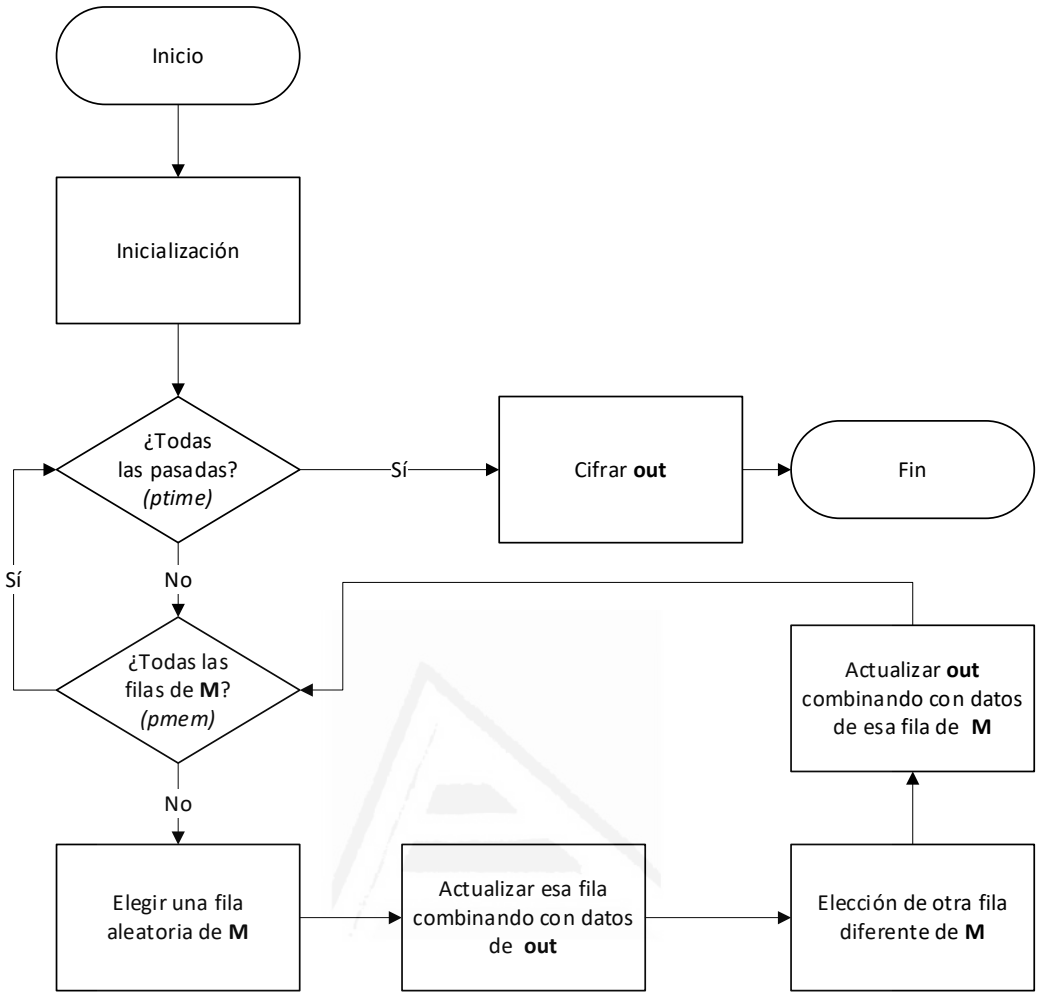


Figura 10. Diagrama de flujo para la salida de AESCTR-i

4.2.2.3 Resultados

Los resultados para la versión AESCTR-i se analizan de forma conjunta con la versión final optimizada (AESCTR-f) en la página 66.

5 Propuesta optimizada final: AESCTR-f

En este capítulo se describe la propuesta principal de este trabajo, una función PBKDF optimizada, a la que denominamos AESCTR-f, capaz de competir con el estado del arte actual tanto en términos de rendimiento como de seguridad.

Esta optimización final mejora aún más el rendimiento respecto a la optimización intermedia (AESCTR-i) mediante la optimización del acceso a memoria y la simplificación del bucle interno de la función de salida.

La etapa de inicialización guarda la misma estructura que para las versiones original e intermedia; existiendo diferencias en la parametrización y la etapa de salida, principalmente.

5.1 Parametrización

En la Tabla 8 se detallan los elementos del algoritmo, algunos de ellos han sido actualizados respecto a las versiones anteriores.

Tabla 8. Elementos de AESCTR-f

M[]:	Matriz de bytes de longitud pmen por plen bytes. Constituye el principal buffer de memoria en el algoritmo.
out[]:	Matriz de plen bytes, contiene el hash de salida.
M64[], out64[]:	Son utilizados en la optimización final (AESCTR-f) para realizar operaciones nativas en 64-bit por razones de rendimiento.
fH():	Función hash criptográfica segura con un tamaño de resumen de 256 bits utilizado para la semilla. En este caso, se emplea el estándar SHA3-256.
fC():	Cifrador por bloque AES-128 estándar, en modo CTR que actúa a modo de PRNG

5.2 Diseño

La principal diferencia de la versión optimizada (AESCTR-f) respecto a la versión intermedia consiste en evitar la escritura en la matriz M[] y eliminar el segundo bucle interno. Además, el acceso a la memoria y las operaciones se realizan en 64 bits, lo que ofrece un rendimiento superior en las arquitecturas más frecuentes.

En la Tabla 9 se describe el pseudocódigo para la función de salida. Se han resaltado las diferencias entre esta optimización final y la versión intermedia. Además, en la Figura 11 se detalla el diagrama de flujo global de AESCTR-f con el objetivo de establecer el contexto de la función de salida dentro del algoritmo PBKDF propuesto.

Tabla 9. Pseudocódigo para la salida de AESCTR-f

```
for t := 0; t < ptime; t++ {
    for m := 0; m < pmem; m++{
        i := (int(out[0:8])%pmem)*plen/8
        for o := 0; o < len (out); o++ {
            Out64[o] -= (M64[i+o] ^out64[o]
        }
    }
}
fC.XORKeyStream(out, out)
```

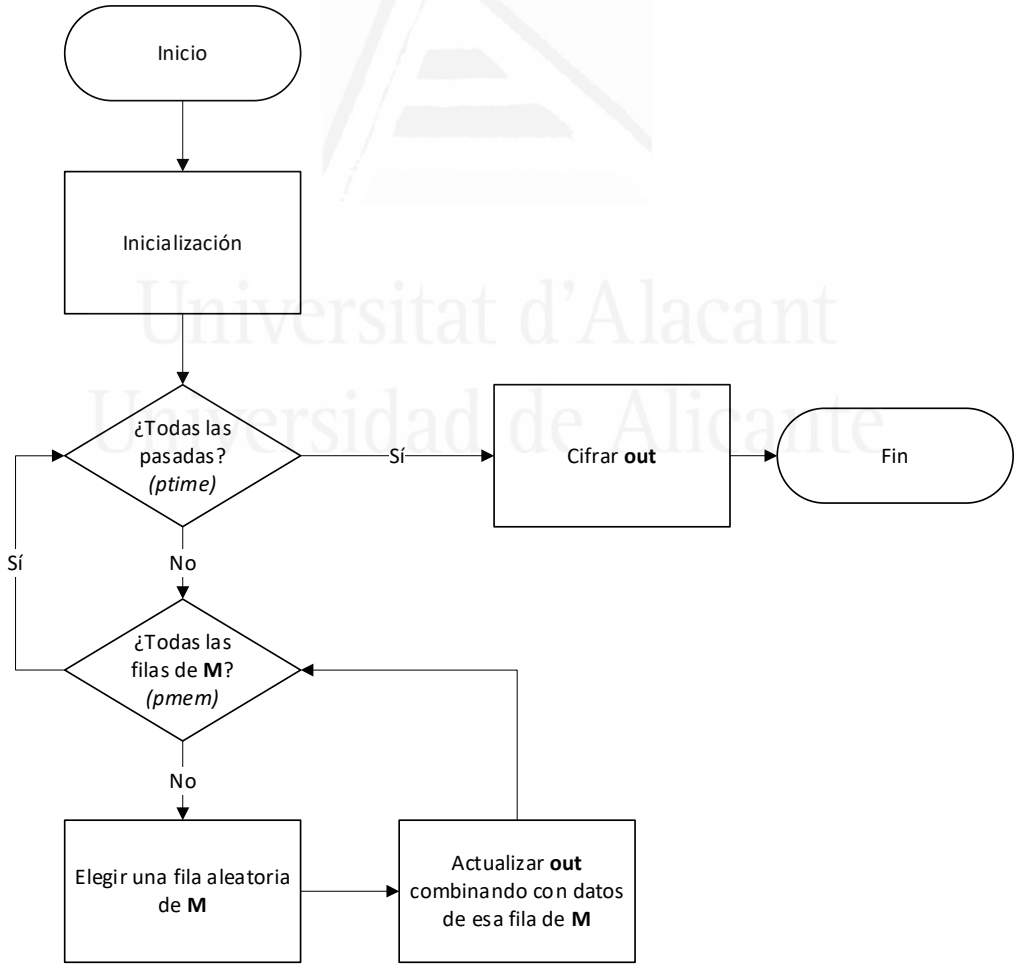


Figura 11. Diagrama de flujo de AESCTR-f

5.3 Resultados

5.3.1 Rendimiento

Tras detallar el diseño, se muestra a continuación el análisis de rendimiento de la propuesta. Para ello, se comparan los tiempos de ejecución de las tres versiones descritas, original (AESCTR-o), optimización intermedia (AESCTR-i) y final (AESCTR-f), al modular los parámetros de complejidad de tiempo (*ptime*) y espacio (*pmem*).

En la Figura 12, se presenta el coste computacional en tiempo de ejecución con escala logarítmica en base 10 de las tres variantes de la propuesta. La modulación del parámetro *pmem* varía de 2^8 a 2^{23} entradas de 32 bytes (valores de *pmem* de 8 a 23 y equivalente a un uso de memoria de 8 KiB a 256 MiB) y se fija el coste temporal con *ptime* = 1. Se detallan los valores precisos en los anexos, página 95.

Se puede ver que la optimización intermedia es una mejora significativa sobre la función original; sin embargo, está claramente superada por la optimización final, que es aproximadamente cinco veces más rápida que la versión original.

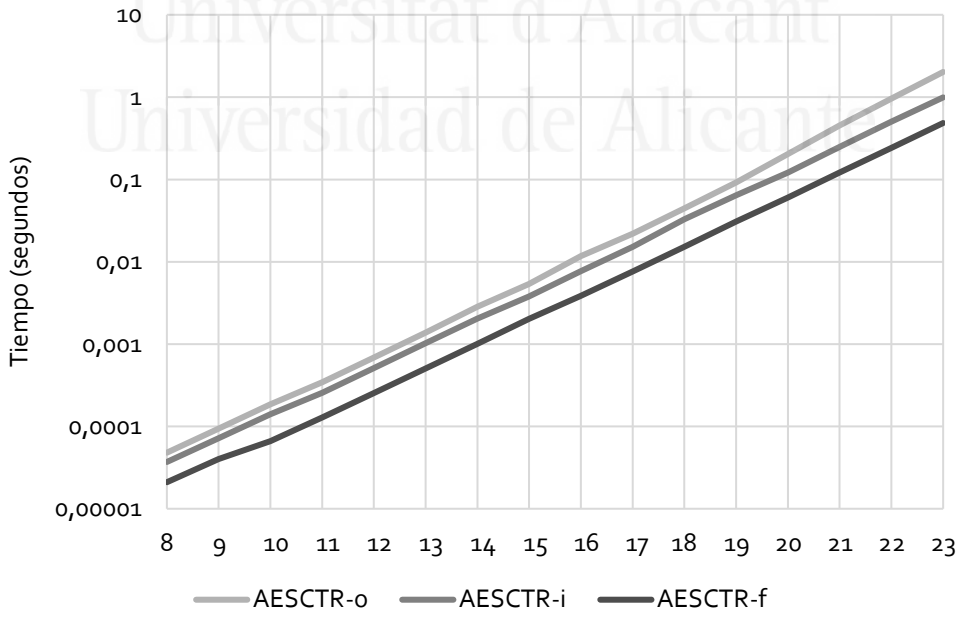


Figura 12. Rendimiento al modular el coste espacial (pmem)

A su vez, en la Figura 13, se muestra el comportamiento (con el tiempo en escala logarítmica de base 10) de las optimizaciones propuestas cuando se modula el parámetro de tiempo. En esta prueba, los valores de *ptime* varían de 1 a 2^{15} pasadas y el uso de la memoria (*pmem*) se mantiene constante en 2^8 entradas de 32 bytes (equivalente a 8KiB de memoria total). En este caso, la diferencia en el rendimiento de la optimización final AESCTR-f es más pronunciada que en el caso del parámetro de memoria. Los valores se incluyen en los anexos, página 96.

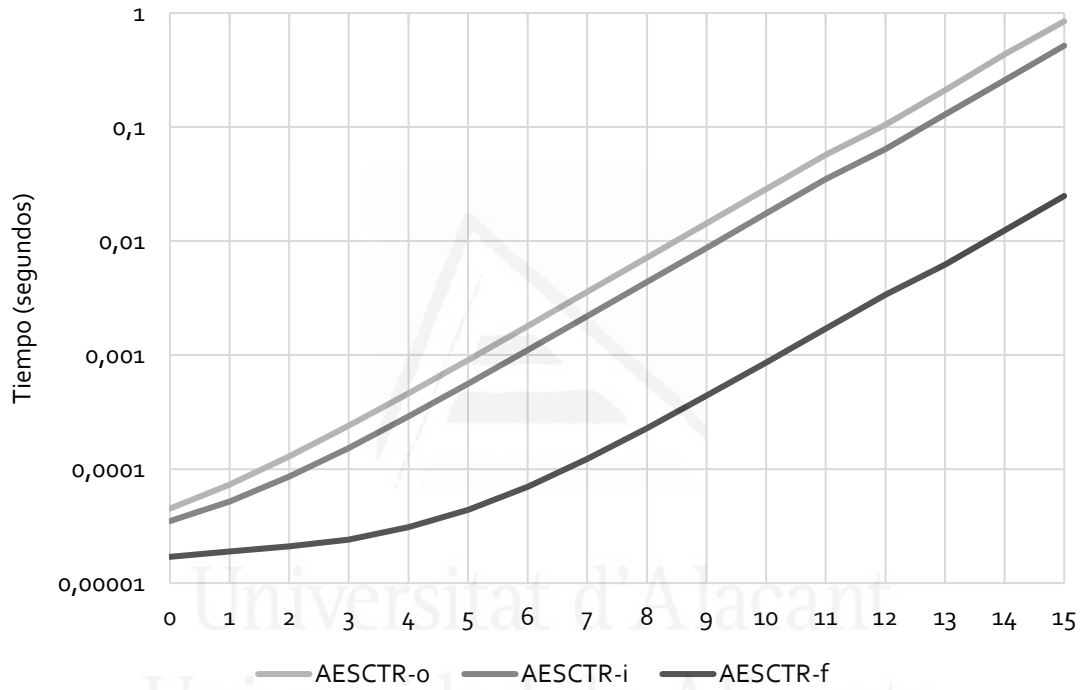


Figura 13. Rendimiento al modular el coste temporal (ptime)

En la Figura 14 se representa el tiempo de ejecución (en escala logarítmica en base 10) cuando ambos parámetros, *pmem* y *ptime*, se modulan simultáneamente en un bucle doble. El bucle interno es *pmem*, que corresponde al número de entradas en el búfer de memoria principal, M[], con valores de 2^8 a 2^{15}, y el bucle externo corresponde a *ptime*, con valores de 1 a 2^7; la cantidad máxima de uso de memoria es de 128MiB y ocurre cuando *pmem* = 2^{15}.

Las formas resultantes de dientes de sierra son esperables en esta disposición de doble bucle. En este punto de referencia combinado, la ganancia de rendimiento lograda con la optimización final es evidente.

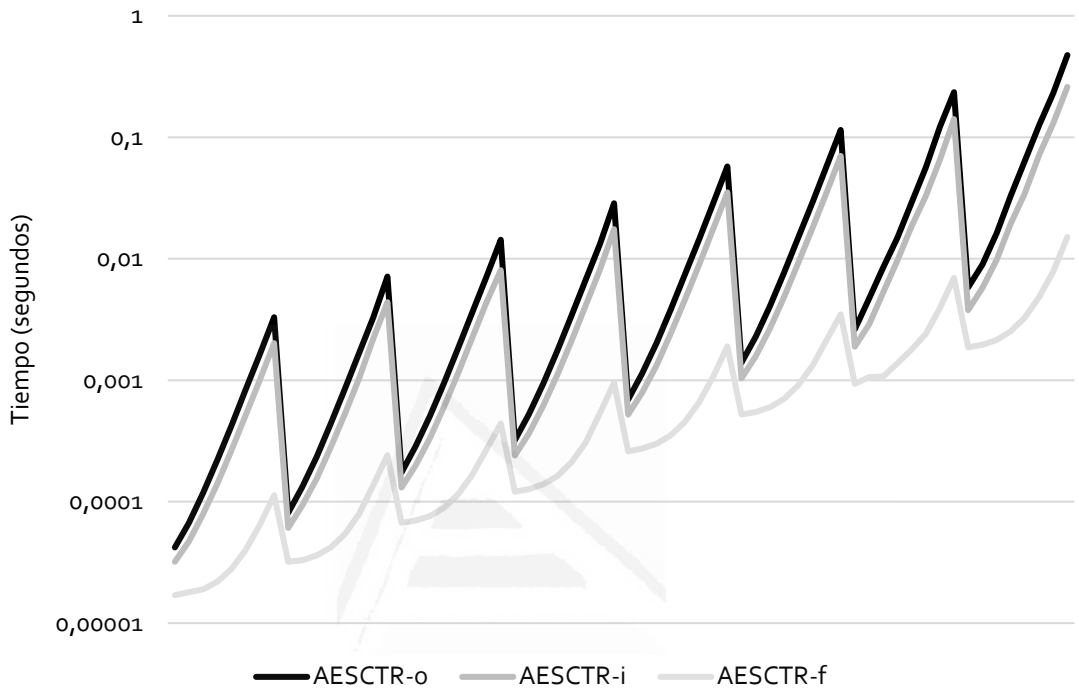


Figura 14. Rendimiento al modular ambos parámetros de forma simultánea

5.3.2 Comparativa

En esta sección se comparan en términos de rendimiento las tres variantes de la función AESCTR con el estado del arte en PBKDF, Scrypt y Argon2.

Como se ha descrito en la sección 4.2.1, Scrypt (véase [7]) es una PBKDF que fue diseñada por Colin Percival en 2009. Se ha empleado para muchos servicios y aplicaciones, actuando como un estándar de facto en los últimos años. También se ha empleado como un algoritmo de prueba de trabajo en algunas implementaciones de *blockchain*.

Como se ha detallado en la sección 3.2.1, Argon2 (véase [91]) se convirtió en el ganador del concurso *Password Hashing Competition* (PHC) en 2015, superando a finalistas como Catena [9],

Lyra2 [12], yescrypt [13] y Makwa [10]. Además, Argon2 está incrementando su presencia en muchos servicios para la realización de autenticación segura de usuarios basada en contraseña.

Es interesante observar que Argon2 se implementó utilizando el conjunto de instrucciones SSE4 de Intel y que la PBKDF propuesta aprovecha las instrucciones primitivas de AES. No obstante, Scrypt no se beneficia directamente de la aceleración de hardware disponible en los procesadores modernos.

En la Figura 15 se muestra el tiempo en escala logarítmica en base 10 según se aumenta el uso de memoria (*pmem* de 8 a 15). Como se puede observar, para cantidades equivalentes de memoria, Scrypt es más lento que las tres versiones de AESCTR propuestas.

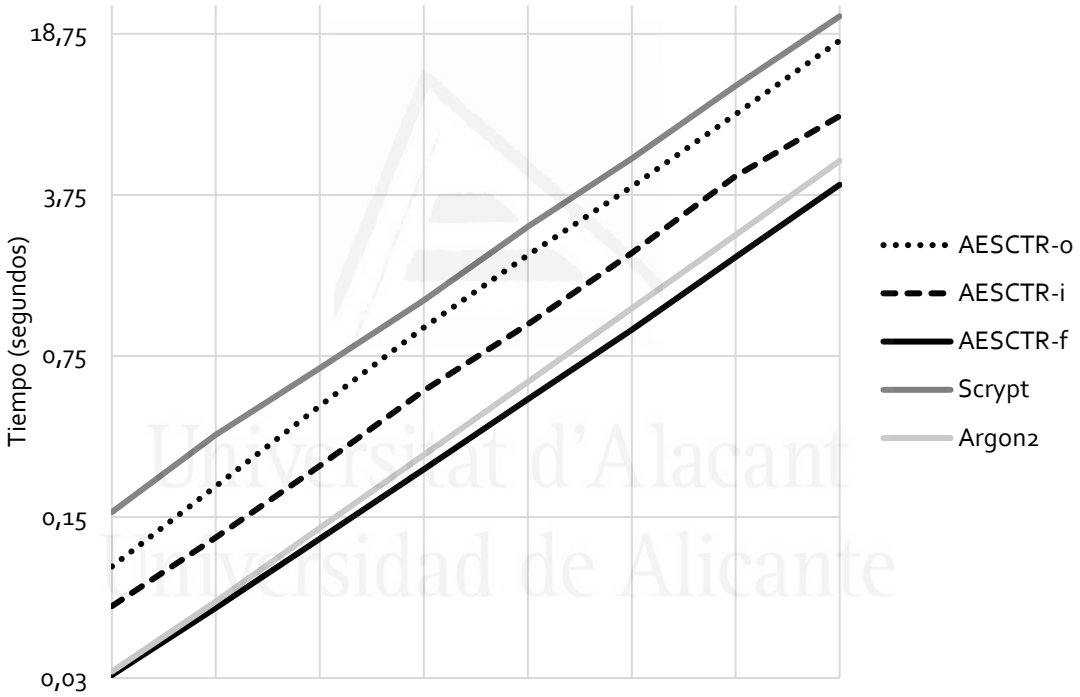


Figura 15. Comparativa con Scrypt y Argon2

Si bien Argon2 es más eficiente que Scrypt, la versión AESCTR-f es ligeramente más rápida que Argon2, como se muestra en detalle en la Tabla 10. Cabe destacar que el incremento de rendimiento (speedup) es mejor al aumentar el uso de memoria. Los valores completos de la comparativa se incluyen en los anexos, página 99.

Tabla 10. Incremento de rendimiento entre AESCTR-f y Argon2

pmem	AESCTR-f	Argon2	Speedup
8	0,03	0,03	1,04
9	0,06	0,06	1,07
10	0,12	0,13	1,11
11	0,24	0,28	1,16
12	0,49	0,58	1,19
13	0,98	1,21	1,24
14	2,02	2,51	1,25
15	4,15	5,27	1,27

5.3.3 Metodología

Como se ha mencionado anteriormente, se ha empleado el lenguaje de programación Go con la versión 1.11.1 (véase [94]) para la implementación de todos los algoritmos probados; Go es una excelente opción para las pruebas de algoritmos criptográficos, ya que es un lenguaje compilado muy eficiente e incluye la mayoría de los estándares y algoritmos de seguridad en su librería estándar.

Todas las pruebas comparativas de esta sección han sido ejecutadas en el mismo computador, un computador de escritorio con una CPU Intel Core i7 (6950X 3.5 GHz y con soporte de aceleración de AES por hardware) y 32 GiB de RAM, con Microsoft Windows 10 (versión 1803). Como longitud para las contraseñas, la sal y la salida se eligió 32 bytes (256 bits), y las pruebas presentadas se ejecutaron 100 veces, para evitar en lo posible la interferencia de procesos externos o tareas.

Para el caso de los puntos de referencia de comparación, las implementaciones oficiales disponibles en los paquetes *golang.org/x/crypto* se utilizaron para las pruebas de Scrypt y Argon2. Teniendo en cuenta que esta implementación de Argon2 soporta la aceleración por hardware a través de las instrucciones SSE4 del procesador, también se usaron los parámetros recomendados de tres pasadas de la memoria y un hilo de ejecución.

Con el objetivo de garantizar una comparativa justa, se eligió una cantidad igual de uso de RAM para cada algoritmo en todas las pruebas de comparación.

Finalmente, basándose en los resultados presentados, se puede considerar que los objetivos planteados en esta investigación se han alcanzado en el contexto que fueron propuestos.

5.4 Análisis de seguridad

El diseño de las funciones AESCTR presentadas, se basa en dos primitivas criptográficas diferentes: un cifrador en bloque en modo CTR que actúa como generador pseudoaleatorio y que proporciona los contenidos iniciales de la memoria y de los buffers de salida; y una función hash criptográfica que procesa la contraseña de usuario más la sal y produce una semilla para este generador pseudoaleatorio.

Se ha empleado el cifrado simétrico AES-128 (véase [95]) en modo CTR, como el generador pseudoaleatorio y SHA3-256 (véase [96]) como la función hash que proporciona la semilla (clave) de 128 bits y el vector de inicialización para AES. Ambas primitivas criptográficas son estándares actuales conocidos y probados en multitud de escenarios independientes. Sin embargo, si fueran consideradas inseguras en el futuro, podrían ser reemplazados por otras alternativas seguras, compatibles con el tamaño y el formato establecido.

Todas las versiones de la función PBKDF propuesta deben considerarse con un nivel de seguridad mínimo de 128 bits contra ataques por fuerza bruta, ya que la salida de la PBKDF en las tres variantes proviene directamente del cifrado simétrico AES-128 y, por lo tanto, se requeriría el criptoanálisis de este cifrado para atacar de forma global a la función propuesta.

6 Conclusiones

En este último capítulo se exponen las conclusiones obtenidas durante el transcurso de la investigación, se aporta un resumen de las contribuciones realizadas y se describen viables líneas futuras de investigación para continuar con el desarrollo de funciones hash criptográficas, tanto para nuevos servicios de hardware como de software.

En la investigación presentada se expone una función de hash de contraseñas basada en el algoritmo AES (Advanced Encryption Standard) en modo contador CTR que aprovecha el soporte de hardware acelerado en la mayoría de los sistemas modernos con el propósito de prevenir los ataques por fuerza bruta que tratan de adivinar o recuperar las contraseñas almacenadas, empleando hardware especializado o unidades de procesamiento gráfico de propósito general.

Se pueden destacar las siguientes conclusiones:

- Los algoritmos propuestos están diseñados, principalmente, como funciones de hashing de contraseñas y de derivación de clave o PBKDF, con la misión principal de autenticación del usuario; proporcionando contramedidas específicas contra atacantes que utilicen unidades de procesamiento gráfico rápidas o hardware personalizado. Las principales ventajas contra estos ataques son el uso de memoria que previene el paralelismo y el hecho de que AES se implementa en hardware en los procesadores modernos.

- Tras el análisis y pruebas de rendimiento de las tres variantes en comparación con funciones como Scrypt y Argon2, que son estándares actuales en términos de funciones de hash de contraseñas, se extrae que la propuesta final optimizada (AESCTR-f) es más rápida que Argon2 para una cantidad igual o equivalente en el uso de memoria, lo que demuestra que AES puede ser un excelente candidato para el diseño de funciones de hashing de contraseñas.

- La seguridad general del diseño de PBKDF propuesto, AESCTR, es equivalente a la del estándar de cifrado AES, ya que se utiliza como una primitiva central en el diseño y la salida final es el resultado directo del cifrado AES.

6.1 Aportaciones

A continuación, se enumeran las principales publicaciones realizadas durante el desarrollo de la investigación del presente trabajo, incluyendo una publicación indexada en el Journal Citation Reports (JCR):

- "AES-CTR as a Password-Hashing Function" [88]. Ponencia en la conferencia internacional *Computational Intelligence and Security in Information Systems* (CISIS, clasificada como CORE B) en 2017; junto con la publicación en la prestigiosa serie de *proceedings* Advances in Intelligent Systems and Computing (AISC) de Springer.

- "Mejora de la seguridad en la autentificación basada en contraseñas mediante un cifrador en bloque" [97]. Ponencia y publicación en la conferencia internacional *Reunión Española sobre Criptología y Seguridad de la Información* (RECSI) en 2018. Está considerada como la conferencia científica de habla hispana de referencia en materia de ciberseguridad.

- "Optimizing a Password Hashing Function with Hardware-Accelerated Symmetric Encryption" [93]. Publicación en 2018 en la prestigiosa revista *Symmetry* de la editorial MDPI. Está indexada dentro del JCR con un factor de impacto que la sitúa en Q2 en el año de publicación.

6.2 Futuras líneas de investigación

Como posibles áreas de investigación futura, destacan las que se describen a continuación.

El uso de ROM en el servidor es una medida de seguridad adicional que implica el empleo de un archivo aleatorio muy grande en el servidor como parte del proceso de hashing. De esta manera, un atacante tendría que conseguir el mismo archivo aleatorio y sería necesaria una gran cantidad de memoria adicional que disuadiría aún más el uso de hardware especializado. La PBKDF propuesta, AESCTR-f, puede ser adaptada para aceptar dicho archivo como parte del algoritmo sin comprometer su rendimiento o seguridad.

La actualización independiente del cliente es la capacidad de cambiar los parámetros de la función PBKDF sin necesidad de que el usuario vuelva a introducir la contraseña. Esta es una característica conveniente en la autenticación de contraseñas, ya que el servidor puede aumentar la seguridad de la PBKDF según sea necesario, pero sin ninguna fricción para los usuarios finales. Esto se puede realizar sin modificar el algoritmo propuesto mediante el procesamiento en varias etapas, pero puede resultar interesante estudiar el uso de otros métodos optimizados.

La descarga del servidor implica delegar parte del cálculo del algoritmo PBKDF al usuario final para que el servidor se vea menos afectado por los requisitos computacionales de la autenticación por contraseña de un gran número de usuarios de forma simultánea. Es, en esencia, una forma de aumentar el paralelismo del servidor y reducir la ventaja que los atacantes pueden tener utilizando unidades de procesamiento gráfico de propósito general u otro hardware especializado. Esto involucra, usualmente, algún tipo de protocolo para compartir la carga computacional entre el servidor y el nodo de usuario de una manera segura.

El paralelismo haciendo uso de múltiples hilos de ejecución también puede estudiarse e incorporarse modificando la optimización final propuesta (AESCTR-f). Esto puede ser útil en situaciones en las que no es posible la descarga computacional del servidor o cuando se desea un mayor paralelismo. Dado que Argon2 permite el paralelismo mediante múltiples hilos, una comparación entre la escalabilidad del rendimiento paralelo de la optimización final y Argon2 podría ser interesante como trabajo futuro.

La implementación especializada en plataformas de hardware, tales como unidades de procesamiento gráfico de propósito general (GP-GPUs) o chips programables (FPGAs), podría ser muy útil para obtener características específicas de rendimiento y optimización adicionales del algoritmo PBKDF propuesto.

Bibliografía

[1] J. Riley, «Tim Berners-Lee», *The Computer Bulletin,* vol. 40, nº 1, pp. 16-17, 1998.

[2] F. Pacheco, Criptografía, Ciudad Autónoma de Buenos Aires:: Fox Andina, 2014.

[3] N. Milosevic, «History of malware», *arXiv: Cryptography and Security,* vol. 1, nº 16, pp. 58-66, 2013.

[4] R. Brewer, «Ransomware attacks», *Network Security archive,* vol. 2016, nº 9, pp. 5-9, 2016.

[5] S. H. Khan, M. A. Akbar, F. Shahzad, M. Farooq y Z. H. Khan, «Secure biometric template generation for multi-factor authentication», *Pattern Recognition,* vol. 48, nº 2, pp. 458-472, 2015.

[6] M. E. Hellman, «A cryptanalytic time-memory trade-off», *IEEE Transactions on Information Theory,* vol. 26, nº 4, pp. 401-406, 1980.

[7] C. Percival, «Stronger Key Derivation via Sequential Memory-Hard Functions», 2009. http://www.tarsnap.com/scrypt/scrypt.pdf

[8] A. Biryukov, D. Dinu y D. Khovratovich, «Argon2: new generation of memory-hard functions for password hashing and other applications» de *IEEE European Symposium on Security and Privacy (EuroSP)*, 2016.

[9] C. Forler, S. Lucks y J. Wenzel, «The Catena Password-Scrambling Framework», Bauhaus-Universitt Weimar, 2015.

[10] T. Pornin, «The MAKWA Password Hashing Function. Version 1.1», Password Hashing Competition, 2015.

[11] D. Mazières, T. J. Sutton y N. Provos, «A Future-Adaptable Password Scheme», *Proceedings of 1999 USENIX Annual Technical Conference,* 1999.

[12] E. Andrade, M. Simplicio, P. Barreto y P. Santos, «Lyra2: efficient password hashing with high security against time-memory trade-offs», *IEEE Transactions on Computers,* vol. PP, nº 99, p. 3096–3108, 2016.

[13] S. Designer, «yescrypt – password hashing scalable beyond bcrypt and scrypt», PHDays, 2014.

[14] K. Moriarty, B. Kaliski y A. Rusch, «PKCS #5: Password-Based Cryptography Specification Version 2.1», 2017. https://rfc-editor.org/info/rfc8018

[15] A. Biryukov, D. Dinu y D. Khovratovich, «Fast and Tradeoff-Resilient Memory-Hard Functions for Cryptocurrencies and Password Hashing», *IACR Cryptology ePrint Archive,* vol. 430, pp. 1-15, 2015.

[16] N. Ferguson, B. Schneier y T. Kohno, Cryptography Engineering: Design Principles and Practical Applications, Wiley, 2010.

[17] J. Daemen y V. Rijmen, The Design of Rijndael: AES — The Advanced Encryption Standard, Springer, 2013.

[18] S. Keller, «NIST-Recommended Random Number Generator Based on ANSI X9. 31 Appendix A. 2.4 Using the 3-Key Triple DES and AES Algorithms», NIST, 2005.

[19] M. J. Lucena Lopez, Criptografía y seguridad en computadores, 2015.

[20] R. Álvarez, Aplicaciones de las matrices por bloques a los criptosistemas de cifrado en flujo, Universidad de Alicante, 2005.

[21] A. Fúster-Sabater, Criptografía, protección de datos y aplicaciones, Madrid: RA-MA, 2012.

[22] C. E. Shannon, «A Mathematical Theory of Communication», Bell System Technical Journal, vol. 27, nº 379, pp. 623-656, 1948.

[23] M. S. Turan, E. Barker, J. Kelsey, K. A. McKay, M. L. Baish y M. Boyle, «Recommendation for the entropy sources used for random bit generation», *NIST Special Publication,* vol. 800, nº 90B, 2018.

[24] A. Saha, N. Manna y S. Mandal, Information Theory, Coding and Cryptography, Pearson, 2013.

[25] J. Hoffstein, J. Pipher y J. H. Silverman, An Introduction to Mathematical Cryptography, Springer, 2014, p. 329.

[26] S. Arora y B. Barak, Computational complexity: a modern approach, Cambridge University Press, 2009.

[27] B. Schneier, «Cryptography: Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)» de *International Workshop on Fast Software Encryption*, 1993.

[28] B. Schneier, J. Kelsey, D. Whiting, D. A. Wagner y C. Hall, «On the Twofish Key Schedule» de *International Workshop on Selected Areas in Cryptography*, pp. 27-42, Springer, 1998.

[29] H. Stevens, «Hans Peter Luhn and the birth of the hashing algorithm», *IEEE Spectrum,* vol. 55, nº 2, pp. 44-49, 2018.

[30] R. Merkle, «Secrecy, authentication, and public key systems», Stanford University, 1979.

[31] J. S. Coron, Y. Dodis, C. Malinaud y P. Puniya, «Merkle-Damgård revisited: How to construct a hash function» de *Annual International Cryptology Conference*, 2005.

[32] R. C. Merkle, «One way hash functions and DES» de *Conference on the Theory and Application of Cryptology*, 1989.

[33] K. Suzuki, D. Tonien, K. Kurosawa y K. Toyota, «Birthday paradox for multi-collisions» de *International Conference on Information Security and Cryptology*, pp. 29-40, Springer, 2006.

[34] D. Boneh, «Twenty Years of attacks on the RSA Cryptosystem», *Notices of the American Mathematical Society,* vol. 46, nº 2, p. 203–213, 1999.

[35] R. L. Rivest, «The ThreeBallot Voting System», *Caltech/MIT Voting Technology Project*, 2006.

[36] L. R. Knudsen, J. E. Mathiassen, F. . Muller y S. S. Thomsen, «Cryptanalysis of MD2», *Journal of Cryptology,* vol. 23, nº 2, pp. 72-90, 2010.

[37] S. S. Thomsen, «Cryptographic Hash Functions», *DTU Library,* 2009.

[38] B. Den Boer y A. Bosselaers, «An attack on the last two rounds of MD4» de *Annual International Cryptology Conference*, 1991.

[39] X. Wang, D. Feng, X. Lai y H. Yu, «Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD», *IACR Cryptology ePrint Archive,* 2004.

[40] Y. Sasaki, Y. Naito, N. Kunihiro y K. Ohta, «Improved Collision Attack on MD5», *IACR Cryptology ePrint Archive,* vol. 105, nº 396, pp. 35-42, 2005.

[41] Google, «Announcing the first SHA1 collision», 2017. https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

[42] B. Preneel, «The First 30 Years of Cryptographic Hash Functions and the NIST SHA-3 Competition», *Lecture Notes in Computer Science,* vol. 5985, nº , pp. 1-14, 2010.

[43] H. Dobbertin, A. Bosselaers y B. Preneel, «RIPEMD-160: A Strengthened Version of RIPEMD», *Lecture Notes in Computer Science,* vol. 1039, pp. 71-82, 1996.

[44] I. Giechaskiel, C. Cremers y K. B. Rasmussen, «When the Crypto in Cryptocurrencies Breaks: Bitcoin Security under Broken Primitives», *IEEE Security & Privacy,* vol. 16, nº 4, pp. 46-56, 2018.

[45] K. Suzuki y K. Kurosawa, «How to find many collisions of 3-pass HAVAL», *IACR Cryptology ePrint Archive,* vol. 2007, nº , pp. 428-443, 2007.

[46] E. Biham, «New Techniques for Cryptanalysis of Hash Functions and Improved Attacks on Snefru» de *International Workshop on Fast Software Encryption*, 2008.

[47] F. Mendel, N. Pramstaller y C. Rechberger, «A (Second) Preimage Attack on the GOST Hash Function», *Lecture Notes in Computer Science,* vol. 5086, nº , pp. 224-234, 2008.

[48] J. Lee y M. Stam, «MJH: a faster alternative to MDC-2», *Designs, Codes and Cryptography,* vol. 76, nº 2, pp. 179-205, 2015.

[49] E. Biham y A. Shamir, «Differential cryptanalysis of feal and N-hash» de *Workshop on the Theory and Application of of Cryptographic Techniques*, 1991.

[50] V. Rijmen, B. V. Rompay, B. Preneel y J. Vandewalle, «Producing Collisions for PANAMA», *Lecture Notes in Computer Science,* vol. 2355, nº , pp. 37-51, 2002.

[51] B. Baldwin, A. W. Byrne, M. Hamilton, N. Hanley, R. P. McEvoy, W. Pan y W. P. Marnane, «FPGA Implementations of SHA-3 Candidates: CubeHash, Grøstl, LANE, Shabal and Spectral Hash», *IACR Cryptology ePrint Archive,* vol. 2009, pp. 783-790, 2009.

[52] R. Álvarez, G. McGuire y A. Zamora, «The Tangle Hash Function», 2008.

[53] T. Ashur y O. Dunkelman, «Linear analysis of reduced-round cubehash», *IACR Cryptology ePrint Archive,* vol. 2010, pp. 462-478, 2011.

[54] J.-P. Aumasson y R. C. W. Phan, «On the cryptanalysis of the hash function Fugue: Partitioning and inside-out distinguishers», *Information Processing Letters,* vol. 111, nº 11, pp. 512-515, 2011.

[55] M. Bellare, K. Pietrzak y P. Rogaway, «Improved security analyses for CBC MACs», *Lecture Notes in Computer Science,* pp. 527-545, 2005.

[56] T. Peyrin, Y. Sasaki y L. Wang, «Generic related-key attacks for HMAC», *IACR Cryptology ePrint Archive,* vol. 2012, nº , pp. 580-597, 2012.

[57] T. Krovetz, «UMAC: Message Authentication Code using Universal Hashing», 2006. http://rfc-editor.org/info/rfc4418

[58] J. Alwen, B. Chen, C. Kamath, V. Kolmogorov, K. Pietrzak y S. Tessaro, «On the Complexity of Scrypt and Proofs of Space in the Parallel Random Oracle Model», *IACR Cryptology ePrint Archive,* vol. 2016, pp. 358-387, 2016.

[59] J. Alwen, B. Chen, K. Pietrzak, L. Reyzin y S. Tessaro, «Scrypt Is Maximally Memory-Hard», *IACR Cryptology ePrint Archive,* vol. 2016, pp. 33-62, 2017.

[60] Z. Tu y C. Xue, «Effect of bifurcation on the interaction between Bitcoin and Litecoin», *Finance Research Letters,* Elsevier, 2018.

[61] NIST, «SHA-3 Project», 2019. http://csrc.nist.gov/groups/ST/hash/sha-3/index.html.

[62] B. Preneel, «The NIST SHA-3 Competition: A Perspective on the Final Year», *Lecture Notes in Computer Science,* vol. 6737, pp. 383-386, 2011.

[63] J.-P. Aumasson, Serious Cryptography, No Starch Press, 2017.

[64] G. Bertoni, J. Daemen, M. Peeters y G. V. Assche, «The road from Panama to Keccak via RadioGatun» de *Dagstuhl Seminar Proceedings*, 2009.

[65] G. Bertoni, J. Daemen, M. Peeters y G. V. Assche, «On the Indifferentiability of the Sponge Construction» de *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2008.

[66] M. J. Dworkin, «SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions,» 2015.

[67] D. J. Bernstein, «Optimization failures in SHA-3 software», 2012.

[68] X. Guo, S. Huang, L. Nazhandali y P. Schaumont, «Fair and comprehensive performance evaluation of 14 second round SHA-3 ASIC implementations» de *The Second SHA-3 Candidate Conference*, 2010.

[69] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, R. Van Keer y B. Viguier, «KangarooTwelve: Fast Hashing Based on Keccak» de *International Conference on Applied Cryptography and Network Security*, 2018.

[70] J. Kelsey, S. Chang y R. Perlner, «SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash», National Institute of Standards and Technology, 2016.

[71] D. J. Bernstein, «ChaCha, a variant of Salsa20» de *Workshop Record of SASC*, 2008.

[72] J.-P. Aumasson, W. Meier, R. Phan y L. Henzen, The Hash Function BLAKE, Springer, 2014.

[73] J.-P. Aumasson, S. Neves, Z. Wilcox-O'Hearn y C. Winnerlein, «BLAKE2: simpler, smaller, fast as MD5» de *International Conference on Applied Cryptography and Network Security*, 2013.

[74] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schlaffer y S. S. Thomsen, «Grostl-a SHA-3 candidate» de *Dagstuhl Seminar Proceedings*, 2009.

[75] H. Wu, «The hash function JH», *Submission to NIST (round 3),* vol. 6, 2011.

[76] Microsoft, «What is PAE, NX, and SSE2 and why does my PC need to support them to run Windows 8?» http://windows.microsoft.com/en-US/windows-8/what-is-pae-nx-sse2

[77] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas y J. Walker, «The Skein hash function family», *Submission to NIST (round 3),* 2010.

[78] N. At, J.-L. Beuchat y I. San, «Compact Implementation of Threefish and Skein on FPGA», *IACR Cryptology ePrint Archive,* vol. 2012, pp. 1-5, 2012.

[79] B. Preneel, R. Govaerts, J. Vandewalle, «Hash functions based on block ciphers: A synthetic approach» de *Annual International Cryptology Conference*, pp. 368-378, Springer, 1993.

[80] D. Khovratovich, I. Nikolić y C. Rechberger, «Rotational Rebound Attacks on Reduced Skein», *Lecture Notes in Computer Science,* vol. 6477, nº , pp. 1-19, Springer, 2010.

[81] J. P. Aumasson, «Password Hashing Competition», 2019. https://password-hashing.net/

[82] D. Khovratovich, D. Dinu, A. Biryukov y S. Josefsson, «The memory-hard Argon2 password hash and proof-of-work function», 2016. https://tools.ietf.org/html/draft-irtf-cfrg-argon2-04

[83] J. Alwen y V. Serbinenko, «High Parallel Complexity Graphs and Memory-Hard Functions», *IACR Cryptology ePrint Archive,* vol. 2014, pp. 595-603, 2015.

[84] J. Alwen y J. Blocki, «Towards practical attacks on argon2i and balloon hashing» de *IEEE European Symposium on Security and Privacy*, 2017.

[85] C. Forler, S. Lucks y J. Wenzel, «The Catena Password Scrambler», *Submission to Password Hashing Competition (PHC),* 2014.

[86] K. Q. Ye, M. Green, N. Sanguansin, L. Beringer, A. Petcher y A. W. Appel, «Verified Correctness and Security of mbedTLS HMAC-DRBG», *arXiv: Cryptography and Security,* pp. 2007-2020, 2017.

[87] R. Kumar, A. Krishna y J. Anitha, «Detection of HTTP based Botnets», *International journal of engineering research and technology,* vol. 3, nº 19, 2018.

[88] R. Álvarez, A. Andrade, I. Santos y A. Zamora, «AES-CTR as a Password-Hashing Function» de *International Joint Conference SOCO'17-CISIS'17-ICEUTE'17*, León, Spain, Springer, pp. 610–617, 2017.

[89] R. Álvarez y A. Zamora, «Using Spritz as a Password-Based Key Derivation Function» de *International Joint Conference SOCO'16-CISIS'16-ICEUTE'16*, Springer, 2016.

[90] M. A. Simplicio, L. C. Almeida, E. R. Andrade, P. C. Dos Santos y P. S. Barreto, «Lyra2: Password Hashing Scheme with improved security against time-memory trade-offs», IACR Cryptology ePrint Archive, 2015.

[91] A. Biryukov, D. Dinu y D. Khovratovich, «Argon2: the memory-hard function for password hashing and other applications. Password Hashing Competition winner», 2016. https://github.com/P-H-C/phc-winner-argon2/blob/master/ argon2-specs.pdf

[92] M. Fleder, M. S. Kester y S. Pillai, «Bitcoin Transaction Graph Analysis», *arXiv: Cryptography and Security,* 2015.

[93] R. Álvarez, A. Andrade y A. Zamora, «Optimizing a Password Hashing Function with Hardware-Accelerated Symmetric Encryption», *Symmetry*, vol. 10, nº 12, p. 705, 2018.

[94] Google, «The Go Programming Language», 2019. https://golang.org

[95] J. Daemen y V. Rijmen, «AES proposal: Rijndael», 1999.

[96] G. Bertoni, J. Daemen, M. Peeters y G. Van Assche, «Cryptographic sponge functions», 2011. http://sponge.noekeon.org

[97] R. Álvarez, A. Andrade y A. Zamora, «Mejora de la seguridad en la autentificación basada en contraseñas mediante un cifrador en bloque», de *Reunión Española sobre Criptología y Seguridad de la Información (RECSI XV)*, Granada, 2018.

Anexo I: código

MAIN.GO

```
package main

import (
	"crypto/rand"
	"fmt"
	"time"

	"golang.org/x/crypto/argon2"
	"golang.org/x/crypto/scrypt"

	"symm/aesctr"
	"symm/qpc"
)

const iters = 100 //número de iteraciones en las pruebas

//main secuencia las pruebas
func main() {

	fmt.Println("\nMemoria:") // prueba con el parámetro de memoria
	for m := uint(0); m < 16; m++ {
		fmt.Printf("%d, %d, %f %f %f\n", m, 0, testAES(m, 0, 1,
false).Seconds(), testAES(m, 0, 2, false).Seconds(), testAES(m, 0, 3,
false).Seconds())
	}

	fmt.Println("\nTiempo:") // prueba con el parámetro de tiempo
	for t := uint(0); t < 16; t++ {
		fmt.Printf("%d, %d, %f %f %f\n", t, 0, testAES(0, t, 1,
false).Seconds(), testAES(0, t, 2, false).Seconds(), testAES(0, t, 3,
false).Seconds())
	}

	fmt.Println("\nCombinados:") // ambos parámetros combinados
	for m := uint(0); m < 8; m++ {
		for t := uint(0); t < 8; t++ {
			fmt.Printf("%d, %d, %f %f %f\n", m, t, testAES(m, t, 1,
false).Seconds(), testAES(m, t, 2, false).Seconds(), testAES(m, t, 3,
false).Seconds())
		}
	}
```

```
	fmt.Println("\nComparativa con Scrypt y Argon2:") // prueba con Scrypt
	for m := uint(0); m < 16; m++ {
		fmt.Printf("%d, %f, %f, %f <-> %f, %f\n", m, testAES(m, 0, 1,
true).Seconds(), testAES(m, 0, 2, true).Seconds(), testAES(m, 0, 3, true).Seconds(),
testScrypt(m, 0).Seconds(), testArgon2(m, 0).Seconds())
	}

}

// prueba la función AES-CTR para unos parámetros concretos
func testAES(mcost, tcost, ver uint, comp bool) (d time.Duration) {
	// obtener un password y salt aleatorios (para probar)
	pass := make([]byte, 32)
	salt := make([]byte, 32)
	rand.Read(pass)
	rand.Read(salt)

	// ajustar los parámetros como exponentes de potencias de 2
	pm := uint64(1) << (mcost + 8)
	pt := uint64(1) << tcost

	if comp { //comparison mode
		pm *= 8
	}

	// hacer iters pruebas y quedarse con el mínimo tiempo
	min := time.Hour
	for i := 0; i < iters; i++ {
		t := qpc.Now()
		switch ver {
		case 1:
			aesctr.Key(pass, salt, 32, int(pm), int(pt))
		case 2:
			aesctr.Key2(pass, salt, 32, int(pm), int(pt))
		case 3:
			aesctr.Key3(pass, salt, 32, int(pm), int(pt))
		default:
			panic("wrong algorithm version")
		}

		d = qpc.Since(t)
		if d < min {
			min = d
		}
	}
	d = min
```

```
	return d
}

// prueba Scrypt
func testScrypt(mcost, tcost uint) (d time.Duration) {
	pass := make([]byte, 32)
	salt := make([]byte, 32)
	rand.Read(pass)
	rand.Read(salt)
	pm := uint64(1) << (mcost + 8)

	min := time.Hour
	for i := 0; i < iters; i++ {
		t := qpc.Now()
		scrypt.Key(pass, salt, int(pm), 8, 1, 32)
		d = qpc.Since(t)
		if d < min {
			min = d
		}
	}
	d = min

	return d
}

// prueba Argon2
func testArgon2(mcost, tcost uint) (d time.Duration) {
	pass := make([]byte, 32)
	salt := make([]byte, 32)
	rand.Read(pass)
	rand.Read(salt)
	pm := uint64(1) << (mcost + 8)
	pm = pm / 4

	min := time.Hour
	for i := 0; i < iters; i++ {
		t := qpc.Now()
		argon2.IDKey(pass, salt, 3, uint32(pm), 1, 32)
		d = qpc.Since(t)
		if d < min {
			min = d
		}
	}
	d = min

	return d
}
```

AESCTR.GO

```
package aesctr

import (
	"crypto/aes"
	"crypto/cipher"
	"encoding/binary"
	"reflect"
	"unsafe"

	"golang.org/x/crypto/sha3"
)


// Key es la versión original publicada en CISIS 17
func Key(pass, salt []byte, plen, pmem, ptime int) ([]byte, error) {

	M := make([]byte, pmem*plen) // tabla de memoria

	// inicialización
	fH := sha3.New256()       // sha3-256 como función hash
	fH.Write(pass)            // se procesa la contraseña
	fH.Write(make([]byte, 1)) // se incluye un byte '0' como separador
	fH.Write(salt)            // se procesa la sal
	seedout := fH.Sum(nil)    // el resumen se utiliza como clave e IV en AES

	// los primeros 16 bytes (128bits) son la clave de AES
	blk, err := aes.NewCipher(seedout[0:16])
	if err != nil {
		return nil, err
	}

	// los últimos 16 bytes son el IV para el modo CTR
	fC := cipher.NewCTR(blk, seedout[16:32])

	// se inicializa la salida cifrándola (pseudoaleatorio)
	out := make([]byte, plen)
	fC.XORKeyStream(out, out)

	// se inicializa la tabla cifrándola (pseudoaleatorio)
	fC.XORKeyStream(M, M)

	// Búcle de proceso / salida
	for t := 0; t < ptime; t++ { // tantas veces como parámetro de tiempo
```

```
		for m := 0; m < pmem; m++ { // tantas veces como filas tiene la matriz
M
	// obtener un índice de fila pseudoaleatorio a partir del estado actual de la
salida
			i := int(binary.LittleEndian.Uint64(out)%uint64(pmem)) * plen
			for o := 0; o < len(out); o++ {
				M[i+o] -= out[o]
			// procesar esa fila restándola con el estado actual de la
			salida
			}
			fC.XORKeyStream(out, M[i:i+plen])
			// cifrar esa fila y ponerla como la salida actual
		}
	}

	// devolver el último estado de la salida
	return out, nil
}

// Key2 es la versión 2, evitando cifrar por fila
func Key2(pass, salt []byte, plen, pmem, ptime int) ([]byte, error) {

	M := make([]byte, pmem*plen) // tabla de memoria

	// inicialización
	fH := sha3.New256()       // sha3-256 como función hash
	fH.Write(pass)            // se procesa la contraseña
	fH.Write(make([]byte, 1)) // se incluye un byte '0' como separador
	fH.Write(salt)            // se procesa la sal
	seedout := fH.Sum(nil)    // el resumen se utiliza como clave e IV en AES

	// los primeros 16 bytes (128bits) son la clave de AES
	blk, err := aes.NewCipher(seedout[0:16])
	if err != nil {
		return nil, err
	}

	// los últimos 16 bytes son el IV para el modo CTR
	fC := cipher.NewCTR(blk, seedout[16:32])

	// se inicializa la salida cifrándola (pseudoaleatorio)
	out := make([]byte, plen)
	fC.XORKeyStream(out, out)

	// se inicializa la tabla cifrándola (pseudoaleatorio)
	fC.XORKeyStream(M, M)
```

```
	// Búcle de proceso / salida
	for t := 0; t < ptime; t++ { // tantas veces como parámetro de tiempo
		for m := 0; m < pmem; m++ { // tantas veces como filas de la matriz M
// obtener un índice de fila pseudoaleatorio a partir del estado actual de la salida
			i := int(binary.LittleEndian.Uint64(out)%uint64(pmem)) * plen
			for o := 0; o < len(out); o++ {
				M[i+o] -= out[o]
// procesar esa fila restándola con el estado actual de la salida
			}
			i = int((i * i) % pmem) // obtener un nuevo índice
			for o := 0; o < len(out); o++ {
				out[o] -= (M[i+o] ^ out[o])
				// combinar la salida con una nueva fila
			}
		}
	}
	fC.XORKeyStream(out, out) // cifrar el último estado de la salida

	return out, nil // devolver el último estado de la salida
}

// Key3 es la versión 3, evitando actualizar M, acceso independiente de datos
func Key3(pass, salt []byte, plen, pmem, ptime int) ([]byte, error) {

	M := make([]byte, pmem*plen) // tabla de memoria

	// inicialización
	fH := sha3.New256()	// sha3-256 como función hash
	fH.Write(pass)		// se procesa la contraseña
	fH.Write(make([]byte, 1)) // se incluye un byte '0' como separador
	fH.Write(salt)		// se procesa la sal
	seedout := fH.Sum(nil)	// el resumen se utiliza como clave e IV en AES

	// los primeros 16 bytes (128bits) son la clave de AES
	blk, err := aes.NewCipher(seedout[0:16])
	if err != nil {
		return nil, err
	}

	// los últimos 16 bytes son el IV para el modo CTR
	fC := cipher.NewCTR(blk, seedout[16:32])

	// se inicializa la salida cifrándola (pseudoaleatorio)
	out := make([]byte, plen)
	fC.XORKeyStream(out, out)

	// se inicializa la tabla cifrándola (pseudoaleatorio)
```

```
	fC.XORKeyStream(M, M)

	//conversión a 64 bits para acelerar el rendimiento
	hm := *(*reflect.SliceHeader)(unsafe.Pointer(&M))
	hm.Len /= 8
	hm.Cap /= 8
	M64 := *(*[]uint64)(unsafe.Pointer(&hm))

	ho := *(*reflect.SliceHeader)(unsafe.Pointer(&out))
	ho.Len /= 8
	ho.Cap /= 8
	out64 := *(*[]uint64)(unsafe.Pointer(&ho))

	//fmt.Println(out64)
	// Búcle de proceso / salida

	for t := 0; t < ptime; t++ { // tantas veces como parámetro de tiempo
// obtener un índice de fila pseudoaleatorio a partir del estado actual de la salida
		i := int(binary.LittleEndian.Uint64(out)%uint64(pmem)) * plen / 8
		for m := 0; m < pmem; m++ {
		// tantas veces como filas tiene la matriz M
			for o := 0; o < len(out64); o++ {
				out64[o] -= (M64[i+o] ^ out64[o])
				// actualizar el estado actual de la salida con esa fila
			}
		}
	}
	fC.XORKeyStream(out, out) // cifrar el último estado de la salida

	return out, nil // devolver el último estado de la salida
}
```

Anexo II: datos

MEMORIA

pmem	AESCTR-o	AESCTR-i	AESCTR-f
0	0.000048	0.000037	0.000021
1	0.000094	0.000072	0.000040
2	0.000185	0.000141	0.000066
3	0.000344	0.000257	0.000128
4	0.000687	0.000511	0.000254
5	0.001380	0.001026	0.000506
6	0.002846	0.002045	0.001011
7	0.005367	0.003804	0.002017
8	0.011712	0.007664	0.003859
9	0.022003	0.015224	0.007608
10	0.044442	0.032927	0.015188
11	0.091701	0.064216	0.030804
12	0.204335	0.121461	0.060323
13	0.453150	0.247314	0.121525
14	0.956059	0.502778	0.240720
15	2.020288	0.991828	0.484539

ptime	AESCTR-o	AESCTR-i	AESCTR-f
0	0.000045	0.000035	0.000017
1	0.000073	0.000052	0.000019
2	0.000129	0.000086	0.000021
3	0.000240	0.000153	0.000024
4	0.000462	0.000289	0.000031
5	0.000907	0.000559	0.000044
6	0.001799	0.001107	0.000070
7	0.003583	0.002195	0.000123
8	0.007143	0.004384	0.000228
9	0.014289	0.008710	0.000440
10	0.028494	0.017449	0.000860
11	0.056927	0.034801	0.001702
12	0.104943	0.064082	0.003387
13	0.210059	0.128487	0.006224
14	0.434485	0.256305	0.012435
15	0.844812	0.516522	0.024847

COMBINADO

pmem	ptime	AESCTR-o	AESCTR-i	AESCTR-f
0	0	0.000042	0.000032	0.000017
0	1	0.000067	0.000047	0.000018
0	2	0.000118	0.000079	0.000019
0	3	0.000221	0.000141	0.000022
0	4	0.000425	0.000266	0.000028
0	5	0.000835	0.000516	0.00004
0	6	0.001653	0.001014	0.000065
0	7	0.003292	0.002019	0.000113
1	0	0.000081	0.000061	0.000032
1	1	0.000132	0.000093	0.000033
1	2	0.000234	0.000155	0.000036
1	3	0.000439	0.000280	0.000042
1	4	0.000848	0.000529	0.000054
1	5	0.001666	0.001028	0.000079
1	6	0.003298	0.002203	0.000137
1	7	0.007139	0.004376	0.000241
2	0	0.000174	0.000131	0.000067
2	1	0.000286	0.000199	0.00007
2	2	0.000508	0.000335	0.000076
2	3	0.000951	0.000605	0.000089
2	4	0.001841	0.001147	0.000115
2	5	0.003622	0.002233	0.000167
2	6	0.007177	0.004407	0.000271
2	7	0.014297	0.008062	0.000440
3	0	0.000321	0.000240	0.000121
3	1	0.000526	0.000365	0.000127
3	2	0.000937	0.000616	0.000139
3	3	0.001759	0.001114	0.000163
3	4	0.003403	0.002109	0.000211
3	5	0.006679	0.004115	0.000306
3	6	0.013258	0.008151	0.000538
3	7	0.028659	0.017525	0.000952
4	0	0.000694	0.000520	0.000260
4	1	0.001141	0.000790	0.000274
4	2	0.002035	0.001330	0.000300
4	3	0.003818	0.002415	0.000351
4	4	0.007380	0.004580	0.000454
4	5	0.014536	0.008951	0.000661
4	6	0.028814	0.017667	0.001074
4	7	0.057345	0.035152	0.001901

5	0	0.001393	0.001036	0.000520
5	1	0.002290	0.001580	0.000547
5	2	0.004081	0.002663	0.000598
5	3	0.007664	0.004844	0.000700
5	4	0.014851	0.009189	0.000907
5	5	0.029185	0.017893	0.001321
5	6	0.057901	0.035157	0.002145
5	7	0.115080	0.070100	0.003485
6	0	0.002626	0.001886	0.000931
6	1	0.004731	0.002888	0.001063
6	2	0.008506	0.005290	0.001076
6	3	0.014847	0.009656	0.001373
6	4	0.028771	0.018338	0.001785
6	5	0.056563	0.032933	0.002405
6	6	0.121656	0.065275	0.003928
6	7	0.235234	0.140787	0.006965
7	0	0.005789	0.003775	0.001865
7	1	0.008981	0.005779	0.001955
7	2	0.016122	0.009791	0.002147
7	3	0.032927	0.019359	0.002528
7	4	0.063899	0.034524	0.003285
7	5	0.125926	0.071678	0.004844
7	6	0.231295	0.130475	0.007842
7	7	0.474521	0.259669	0.015118

pmem	AESCTR-o	AESCTR-i	AESCTR-f	Scrypt	Argon2
8	0.091424	0.061542	0.030890	0.157464	0.031993
9	0.203874	0.122256	0.060410	0.341424	0.064525
10	0.455151	0.250519	0.120747	0.664289	0.134489
11	0.996637	0.531445	0.241179	1.309539	0.279122
12	2.050638	1.030148	0.486883	2.727834	0.577628
13	4.072673	2.099638	0.975559	5.399169	1.206814
14	8.388755	4.526427	2.015799	11.148730	2.512026
15	17.524090	8.249468	4.151492	22.349150	5.267642