UNIVERSIDAD DE GRANADA

Criptosistemas en aplicaciones de mensajería

Trabajo de fin de grado en Ingeniería Informática y Matemáticas

Autor
Luis Tormo Fabios

Director
Pedro A. García Sánchez

—

Granada, mes de 201

Índice general

Capítulo 1

Introducción

Vivimos en una época en la que no se pueden concebir las relaciones sociales sin pensar en las redes sociales y en particular las aplicaciones de mensajería. Estas nos permiten conectarnos unos con otros, independientemente de las barreras físicas. Hay aplicaciones como *WhatsApp*, *Facebook Messenger* o *Telegram* que tienen 2.000, 931 y 700 millones de usuarios respectivamente, lo que supone un porcentaje significativo de la población mundial que usa aplicaciones de mensajería.
Debido a esta enorme cantidad de usuarios, las aplicaciones tienen que garantizar su seguridad y la privacidad. Es por esto que la criptografía ha cobrado un papel fundamental en la actualidad, ya que las herramientas que ofrece son las que permiten garantizar dicha seguridad y privacidad de los usuarios de las aplicaciones.

1.1. Contexto histórico

Las aplicaciones de mensajería aparecieron en la década de 1970. Una de las primeras fue el sistema *PLATO*, este era una aplicación de asistencia para la computadora basada en un sistema informático de tiempo compartido por usuarios y programadores. Fue diseñada por Bitzer con la finalidad de hacer realidad el objetivo de educar por el ordenador y entre una de sus funcionalidades había un chat para que los usuarios se comunicaran entre sí de manera local.
Entre las décadas de 1980 y 1990 apareció la aplicación *TALK*, esta fue diseñada para dispositivos con sistema operativos basados en *UNIX/LINUX*. Esta aplicación permitía enviar mensajes entre usuarios a través de Internet. Si bien al principio sólo permitía comunicarse entre usuarios que estuvieran conectados al mismo dispositivo, aunque luego se amplió la funcionalidad permitiendo el envío de mensajes entre usuarios de otros sistemas.

Hasta 1996 no apareció una aplicación de mensajería que se pudiera usar en otros dispositivos con distintos sistemas operativos. Esta fue *ICQ* y supuso un antes y un después, ya que fue la primera en abarcar tantos usuarios y además añadió nuevas funcionalidades. En su momento de mayor popularidad alcanzó los 38 millones de usuarios, permitiendo atisbar el potencial de las aplicaciones de mensajería como medio de comunicación. Esta aplicación añadía nuevas funcionalidades como eran un perfil de usuario personalizable, estado de conexión, emoticonos, transferencia de contactos, transferencia de archivos y chat grupales que fueron adoptadas por las nuevas aplicaciones de mensajería, manteniéndose muchas de ellas hasta hoy en día.
A partir de esta empezaron a surgir nuevas aplicaciones de mensajería con mayor frecuencia, estas aplicaciones usaban cada una un protocolo distinto por lo que se llevó a los usuarios a tener distintos clientes para cada aplicación. Algunas de las aplicaciones más populares que aparecieron en esta época fueron *MSN Messenger* y *AIM*. La más popular fue *AIM* que en 2006 tenía el control del 52 por ciento del mercado de las aplicaciones de mensajería. *MSN* necesitó más años para ser más popular y hasta 2005 no alcanzó su mayor pico, llegando a atraer alrededor de 330 millones de usuarios activos cada mes. Lo hizo bajo el nombre de *Windows Live Messenger*.
Para compensar el creciente número de protocolos surgieron aplicaciones multiclientes que permitían soportar varios de estos protocolos, algunas de estas fueron *Pidgin* o *Trillian*. Ambas aplicaciones permitían comunicarse usando protocolos como *MSN*, *MySpaceIM*, *XMPP/Jabber(Google Talk, Facebook Messenger)* y *Yahoo!* entre otros.
A la vez se popularizaron las videollamadas, por lo que aparecieron nuevas aplicaciones para aprovechar el nuevo nicho. Una de las primeras en aparecer fue *Microsoft NetMeeting* aunque poco después apareció *Skype* y se apropió de la mayoría de los usuarios de esta.
En 2010 los desarrolladores cambiaron de plataforma y dejaron de desarrollar aplicaciones de mensajería para ordenador para centrarse en los *Smartphones*. Aparecieron aplicaciones como *WhatsApp, Telegram* y *Facebook Messenger* que como hemos visto al principio del capítulo, son fundamentales hoy en día y abarcan miles de millones de usuarios.

1.2. Descripción del problema

En esta memoria el problema que he abordado ha sido el desarrollo de un estudio teórico de los criptosistemas que utilizan las aplicaciones de mensajería más utilizadas en la actualidad. Para ello he desarrollado de manera rigurosa las herramientas matemáticas e informáticas que utilizan

las aplicaciones de mensajería. Este desarrollo abarca una introducción a la criptografía simétrica y asimétrica, así como teoría de cuerpos finitos necesarios para entender adecuadamente las herramientas. Un desarrollo en profundidad de los cifrados de bloque y del cifrado *AES* en particular, una explicación exhaustiva de *RSA* y del *Problema del Logaritmo discreto* y como resultado de este, el intercambio de claves *Diffie-Hellman*. Posteriormente, he introducido la teoría de Curvas Elípticas necesaria para entender el análogo de las herramientas anteriores utilizando este cuerpo. Después he introducido las *funciones hash*, como construirlas usando la construcción de *Merkle-Damgård* y las familias de funciones más utilizadas en las aplicaciones de mensajería actuales. A continuación he realizado una descripción práctica de cómo incorporan las aplicaciones de mensajería más populares los criptosistemas vistos anteriormente. Por último he desarrollado una aplicación de mensajería en la cual he aplicado lo visto previamente en la memoria *Desarrollar esto más extensamente cuando tenga la aplicación.*

1.3. Técnicas utilizadas

Para poder llevar a cabo esta memoria he necesitado de diversas herramientas matemáticas e informáticas que he ido viendo a lo largo de mi paso por los grados de matemáticas e informática.
Las herramientas matemáticas utilizadas han sido las siguientes.

- Teoría de números: Función phi de Euler, Teorema Chino del Resto, Teorema de Euler y Teorema pequeño de Fermat.
- Teoría de cuerpos: Cuerpos finitos.
- Teoría de grupos: El problema de Logaritmo Discreto.
- Teoría de Galois: Resultados de Cuerpos finitos.
- Geometría lineal: Transformaciones lineales.
- Geometría algebraica: Curvas elípticas y resultados de estas.

Y las herramientas informáticas

- Criptografía: Criptosistema, criptografía simétrica, criptografía asimétrica, funciones hash, firma digital e intercambio de claves *Diffie-Hellman*.
- Algorítmica: Análisis de algoritmos, en casi todas los capítulos de la memoria aparece algún algoritmo y un análisis de este es necesario para su correcta comprensión.

Capítulo 2

Introducción a la Criptografía

En este capítulo voy a introducir los objetivos y ataques de un criptosistema, los criptosistemas simétricos y los distintos modos de funcionamiento de estos, así como los criptosistemas asimétricos y el uso de estos criptosistemas en las aplicaciones de mensajería .

2.1. Objetivos de un criptosistema y posibles ataques

Todo criptosistema se construye con la finalidad de cumplir una serie de objetivos así como de protegerse de unos ataques. A continuación voy explicar de cuales se trata. La información de este apartado ha sido obtenida de [23].

2.1.1. Objetivos

Los objetivos que tiene que cumplir un criptosistema son los siguientes.

Confidencialidad. La información solo puede ser accesible por las entidades autorizadas.

Integridad. La información no ha sido alterada en el envío.

Autenticidad. La información proviene de quién afirma haberla enviado.

No repudio. El emisario de una información no puede negar haber realizado tal envío.

2.1.2. Ataques

Para hablar de los ataques supondremos que se sigue el principio de *Kerckhoffs*, el cual establece que el adversario conoce todos los detalles del criptosistema excepto la clave empleada.
Los posibles ataques son:

Criptograma. El adversario conoce el criptograma, es decir, el mensaje cifrado o un fragmento de este.

Mensaje Conocido. El atacante conoce parejas mensaje/criptograma cifradas con una misma clave.

Mensaje escogido. El atacante puede generar criptogramas para mensajes de su elección. Una vez obtenidas dichas parejas, trata de averiguar el mensaje correspondiente a un criptograma desconocido.

Mensaje escogido-adaptativo. El atacante no solo puede generar pareas mensaje/criptograma a su elección, sino que puede hacerlo tantas veces como quiera realizando los análisis que considere oportunos.

Criptograma escogido y escogido-adaptativo. Similar a los anteriores pero partiendo del criptograma, teniendo acceso a descifrar los criptogramas que desee, inicialmente o a lo largo del proceso. Lo que se busca en este ataque es la clave.

Una vez vistos los objetivos que tienen que cumplir los criptosistemas y los posibles ataques de los que pueden ser objeto, voy a explicar el uso de los criptosistemas simétricos y asimétricos en las aplicaciones de mensajería.

2.2. Criptosistemas simétricos y asimétricos en las aplicaciones de mensajería

Los criptosistemas simétricos y asimétricos conforman un elemento fundamental en las aplicaciones de mensajería. Criptosistemas de ambas familias se usan de manera conjunta para garantizar la confidencialidad, integridad, autenticidad y no repudio de los mensajes.
Los criptosistemas simétricos son utilizados para cifrar los mensajes. Esto es debido a su velocidad de cifrado, su uso reducido de recursos y su mejor manejo de grandes cantidades de datos. Tienen el defecto de que si la clave es interceptada, el criptosistema es vulnerado y se pierde tanto la confidencialidad como la autenticidad de los mensajes. Para evitar esto se suele complementar con métodos seguros para el intercambio de la clave

como puede ser el *intercambio de claves Diffie-Hellman*.
Los criptosistemas asimétricos son muy utilizados para la firma y autentificación de los mensajes, garantizando de esta manera la seguridad de la aplicación y se complementan con cifrados simétricos a la hora de cifrar los mensajes para garantizar de esta forma una eficiencia mucho mayor. Ya que uno de los principales problemas que tienen es su complejidad algorítmica a la hora de cifrar y descifrar los mensajes.

2.3. Criptosistema simétrico

Un criptosistema simétrico es un criptosistema en el cual se utiliza una sola clave para cifrar y descifrar un mensaje o es necesario conocer la clave secreta para desencriptar un mensaje. La importancia para garantizar la seguridad de los criptosistemas simétricos reside en el secreto de la clave, mientras que el conocer el algoritmo utilizado no es tan importante como medida de seguridad. Es decir, lo importante es que el atacante no conozca la clave, mientras que conozca el algoritmo usado no lo es tanto. La información ha sido obtenida de [23].
Un criptosistema simétrico está formado por:

- $\mathcal{M}$ el conjunto de los mensajes, elementos candidatos a ser encriptados.
- $\mathcal{C}$ el conjunto de los criptogramas o mensajes obtenido después del proceso de encriptar.
- $\mathcal{K} \subseteq \mathcal{K}_p \times \mathcal{K}_s$ el espacio de las claves, elementos que se utilizan para encriptar y desencriptar los mensajes.

Un criptosistema simétrico viene definido por dos aplicaciones

$$E : \mathcal{K}_p \times \mathcal{M} \to \mathcal{C},$$

$$\mathcal{D} : \mathcal{K}_s \times \mathcal{C} \to \mathcal{M}.$$

tales que para cualquier clave $k_p \in \mathcal{K}_p$, existe una clave k_s de manera que dato cualquier mensaje $m \in \mathcal{M}$,

$$\mathcal{D}(k_s, E(k_p, m)) = m.$$

Fijada la clave $k_p \in \mathcal{K}_p$ y su correspondiente $k_s \in \mathcal{K}_s$ se definen las funciones de cifrado y descifrado como:

$$E_{k_p} : \mathcal{M} \to \mathcal{C},$$
$$E_{k_p}(m) = E(k_p, m),$$

$$D_{k_p} : \mathcal{C} \to \mathcal{M},$$
$$D_{k_s}(c) = D(k_s, c),$$

2.3.1. Cifrados de bloque

A continuación voy a introducir los cifrados de bloque, ya que estos son fundamentales a la hora de cifrar los mensajes en las aplicaciones de mensajería debido a su eficiencia. El cifrado de bloque más utilizado actualmente es el cifrado **Rindael AES**. La información ha sido obtenida de [23].
Los cifrados de bloque son criptosistemas de clave simétrica en los que la longitud de los bloques y claves es fija.
Este criptosistema se define

$$E : \mathbb{B}^K \times \mathbb{B}^N \to \mathbb{B}^N,$$

$$D : \mathbb{B}^K \times \mathbb{B}^N \to \mathbb{B}^N,$$

donde N es el tamaño del bloque y K es el tamaño de la clave.
Los cifrados tienen distintos modos de operación los cuales dependen solo del tamaño del bloque. Estos modos permiten garantizar la confidencialidad de los mensajes, si bien, no garantizan su integridad. La información para describir los modos la he complementado con [19].
Los distintos modos usados en los cifrados de bloque son:

- **Electronic CodeBook**
  Modo en el cual para una clave dada, se le asigna un bloque de texto fijo cifrado por cada bloque de texto plano. Los pasos que se siguen para encriptar y desencriptar son:
  - ***Cifrado ECB***
    Dividimos m en $m_{[1]} \dots m_{[l]}$ con $m_{[i]} \in \mathbb{B}^N$
    Para $i \in \{1, \dots, l\}$ hacer
    $c_{[i]} = E_k(m_{[i]})$
    Devolvemos $c_{[1]} \dots c_{[l]}$
  - ***Descifrado ECB***
    Dividimos c en $c_{[1]} \dots c_{[l]}$ con $c_{[i]} \in \mathbb{B}^N$

Para $i \in \{1, \ldots, l\}$ hacer
$m_{[i]} = D_k(c_{[i]})$
Devolvemos $m_{[1]} \ldots m_{[l]}$

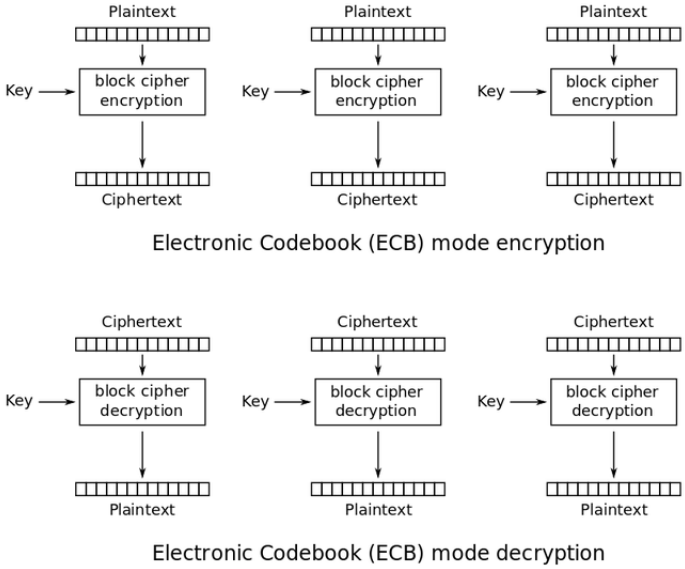


Figura 2.1: Esquema del cifrado y descifrado del modo ECB [2].

- **Cipher-Block Chaining**
  En este modo se combina los bloques de texto plano con los bloques de texto cifrados anteriormente. Para cifrar el primer bloque será necesario un bloque inicial, $c_{[0]}$, el cual no tiene necesariamente que ser secreto. Los pasos seguidos para encriptar y desencriptar son:

  - ***Cifrado CBC***

    $c_{[0]} \in \mathbb{B}^*$
    Dividimos m en $m_{[1]} \ldots m_{[l]}$ con $m_{[i]} \in \mathbb{B}^N$
    Para $i \in \{1, \ldots, l\}$ hacer
    $c_{[i]} = E_k(m_{[i]} \oplus c_{[i-1]})$
    Devolvemos $c_{[1]} \ldots c_{[l]}$
  - ***Descifrado CBC***

    Dividimos c en $c_{[0]} \ldots c_{[l]}$ con $c_{[i]} \in \mathbb{B}^N$
    Para $i \in \{1, \ldots, l\}$ hacer
    $m_{[i]} = D_k(c_{[i]}) \oplus c_{[i]}$
    Devolvemos $m_{[1]} \ldots m_{[l]}$

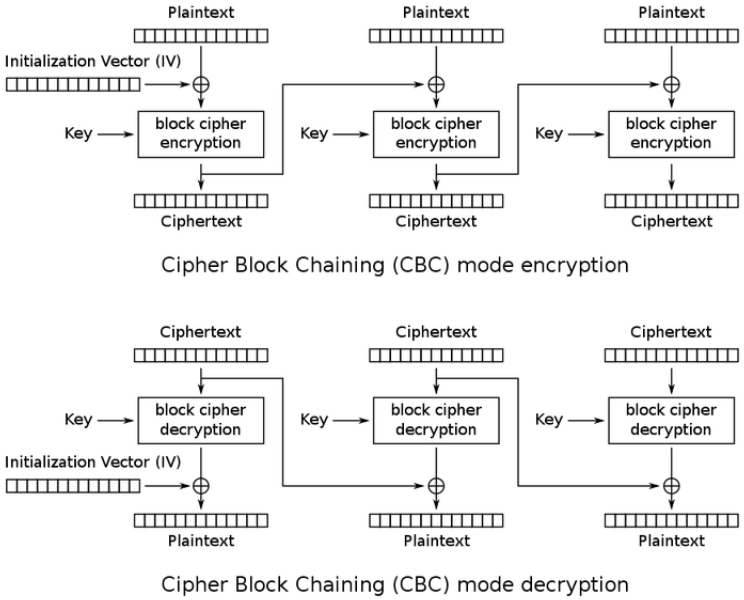


Figura 2.2: Esquema del cifrado y descifrado del modo CBC [2].

- **Cipher FeedBack**
  Modo en el cual se combina cada bloque de texto plano del mensaje consigo mismo encriptado, los pasos que se siguen son:

  - ***Cifrado CFB***

    $x_{[0]} \in \mathbb{B}^r$
    Dividimos m en $m_{[1]} \dots m_{[l]}$ con $m_{[i]} \in \mathbb{B}^N$
    Para $i \in \{1, \dots, l\}$ hacer
    $c_{[i]} = m_{[i]} \oplus msb_r(E_k(x_{[i]}))$
    $x_{[i+1]} = lsb_{N-r}(x_i)||c_{[i]}$
    Devolvemos $c_{[1]} \dots c_{[l]}$

  - ***Descifrado CFB***

    Dividimos c en $c_{[1]} \dots c_{[l]}$ con $c_{[i]} \in \mathbb{B}^r$
    Para $i \in \{1, \dots, l\}$ hacer
    $m_{[i]} = c_{[i]} \oplus msb_r(E_k(x_{[i]}))$
    $x_{[i+1]} = lsb_{N-r}(x_i)||c_{[i]}$
    Devolvemos $m_{[1]} \dots m_{[l]}$

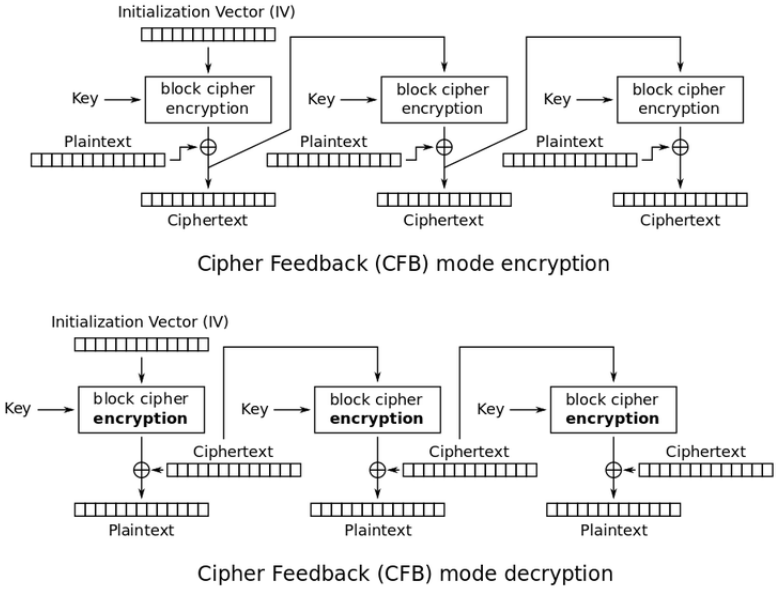


Figura 2.3: Esquema del cifrado y descifrado del modo CFB [2].

- **Output FeedBack**
  Modo en el cual se parte de un bloque inicial $x_{[0]}$ único y secreto. En cada iteración se encripta este y se combina con un bloque del mensaje sin cifrar de manera recursiva. Los pasos seguidos para encriptar y desencriptar son:

  - ***Cifrado OFB***
    $x_{[0]} \in \mathbb{B}^N$
    Dividimos m en $m_{[1]} \dots m_{[l]}$ con $m_{[i]} \in \mathbb{B}^N$
    Para $i \in \{1, \dots, l\}$ hacer
    $x_{[i]} = E_k(x_{[i-1]})$
    $c_{[i]} = m_{[i]} \oplus x_{[i]}$
    Devolvemos $c_{[1]} \dots c_{[l]}$
  - ***Descifrado OFB***
    Dividimos c en $c_{[1]} \dots c_{[l]}$ con $c_{[i]} \in \mathbb{B}^N$
    Para $i \in \{1, \dots, l\}$ hacer
    $x_{[i]} = E_k(x_{[i-1]})$
    $m_{[i]} = c_{[i]} \oplus x_{[i]}$
    Devolvemos $m_{[1]} \dots m_{[l]}$

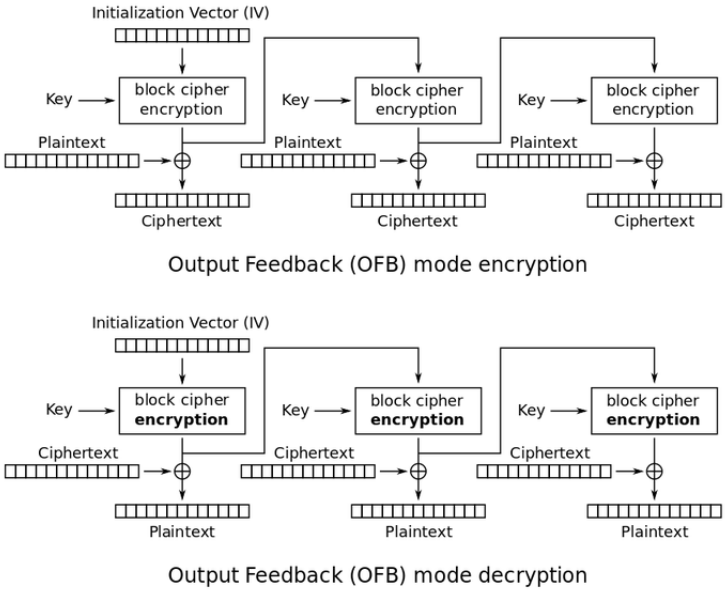


Figura 2.4: Esquema del cifrado y descifrado del modo OFB [2].

- **Galois Counter Mode (GCM)**
  Modo en el cual se usa un función hash universal sobre un cuerpo de Galois binario que provee de una autentica encriptación además de una método de validar los mensajes. La información de este modo ha sido obtenida de [26]. Para cifrar en este modo partimos de una entrada con 4 elementos.

  - K que es una clave secreta.
  - IV vector inicial que puede tener un número de bits entre 1 y 2^{64}.
  - M texto plano que puede tener un número de bits contenido entre 0 y $2^{39} - 256$.
  - AAD datos de autentificación adicionales, *Additional Authenticated Data*, que puede tener un tamaño entre 0 y 2^{64} bits.

  Y en la salida devuelve C que es el texto encriptado y una autentificación T.
  Una vez visto la entrada y la salida, el algoritmo de cifrado y descifrado quedarían como sigue.

- ***Cifrado GCM***

  $H = E_k(0^{128})$
  Si $len(IV) = 96$
  $\quad Y_{[0]} = IV||0^{31}1$
  si no
  $\quad Y_{[0]} = GHASH(H, \{\}, IV)$
  Para $i \in \{1, \ldots, n-1\}$ hacer
  $\quad Y_{[i]} = incr(Y_{[i-1]})$
  $\quad c_{[i]} = m_{[i]} \oplus E(Y_{[i]})$
  $Y_{[n]} = incr(Y_{[n-1]})$
  $c_{[n]} = m_{[n]} \oplus MSB_u(E(Y_{[n]}))$
  $T = MSB_t(GHASH(H, A, C) \oplus E(Y_0))$
  Devolvemos $c = c_{[1]}, \ldots, c_{[n]}$ y T

  Donde $A = A_1, \ldots, A_m$, $incr(F||I) = F||(I + 1 \mod 2^{32})$ y la función $GHASH(H, A, C)$ equivale a calcular X_{m+n+1}. X_i es una variable que se calcula como

$$X_i = \begin{cases} 0 & \text{si } i = 0, \\ (X_{i-1} \oplus A_i) \cdot H & \text{si } i \in \{1, \ldots, m-1\}, \\ (X_{m-1} \oplus (A_m||0^{128-v})) \cdot H & \text{si } i = m, \\ (X_{i-1} \oplus C_i) \cdot H & \text{si } i \in \{m+1, \ldots, m+n-1\}, \\ (X_{m+n-1} \oplus (C_m||0^{128-u})) \cdot H & \text{si } i = m+n, \\ (X_{m+n} \oplus (len(A)||len(C))) \cdot H & \text{si } i = m+n+1. \end{cases}$$

- ***Descifrado GCM***

  $H = E_k(0^{128})$
  Si $len(IV) = 96$
  $\quad Y_{[0]} = IV||0^{31}1$
  si no
  $\quad Y_{[0]} = GHASH(H, \{\}, IV)$
  $T' = MSB_t(GHASH(H, A, C) \oplus (Y_0))$
  Para $i \in \{1, \ldots, n-1\}$ hacer
  $\quad Y_{[i]} = incr(Y_{[i-1]})$
  $\quad m_{[i]} = c_{[i]} \oplus E(Y_{[i]})$
  $Y_{[n]} = incr(Y_{[n-1]})$
  $m_{[n]} = c_{[n]} \oplus MSB_u(E(Y_{[n]}))$
  Devolvemos $m = m_{[1]}, \ldots, m_{[n]}$ y T'

  Si T y T' coinciden entonces se devuelve el texto descifrado. Si no coinciden, entonces el texto no se devuelve ya que esto implicaría que el mensaje ha sido manipulado.

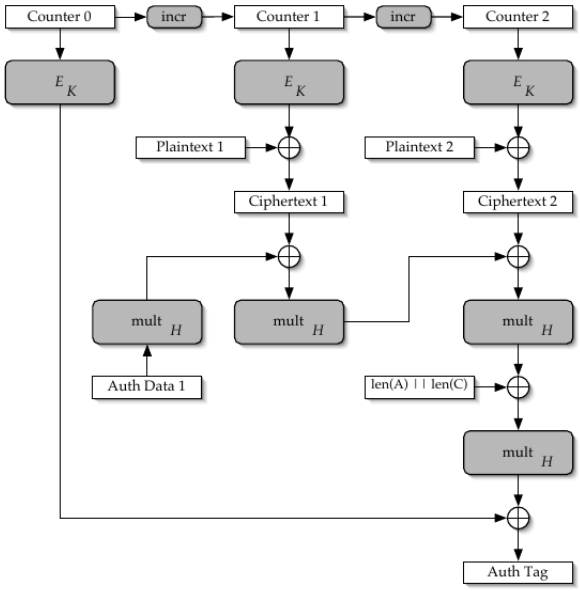


Figura 2.5: Esquema del cifrado y descifrado del modo GCM [26].

Como podemos ver, el modo más sencillo es ECB ya que lo único que hace es fragmentar el mensaje en bloques y encriptar individualmente cada bloque. En CBC, CFB, OFB y GCM se parte de un bloque inicial y se generan bloques nuevos de manera recursiva operando con ellos de manera distinta en función de cada modo.
En CBC se realiza la operación $\oplus$ de cada bloque generado encriptándose el bloque cifrado previo a esta con un bloque del mensaje, los nuevos bloques son los resultados de la operación anterior.
En CFB se coge el bit menos significativo resultante de encriptar el bloque generado previo y se hace la operación $\oplus$ con cada bloque del mensaje. Para generar un nuevo bloque se combina el mensaje cifrado previo con el bit menos significativo del conjunto de bits $N - r$ del bloque generado anterior con la operación $||$.
En OFB se realiza la operación $\oplus$ de el resultado de encriptar el bloque generado previo con un bloque del mensaje.
Y en GCM lo que se hace es fragmentar el mensaje y operar con los fragmentos de manera recursiva usando una función hash universal ($GHASH$) con el primer bloque, además se realiza una operación paralela que se almacena en T para certificar la integridad del mensaje.
Actualmente el más utilizado en las aplicaciones de mensajería es el modo CBC. Esto es debido a que es relativamente fácil de implementar y además permite encriptar en paralelo. Si bien, está empezando a utilizarse el modo GCM ya que implementado en hardware permite unas velocidades muy altas

de encriptado llegando incluso a poder encriptar 10 GB por segundo. Un ejemplo de ello es la aplicación Line, que en su segunda versión incorpora este modo.

2.4. Criptosistema asimétrico

Un criptosistema asimétrico es un criptosistema en el cual se utilizan dos claves, una para cifrar el mensaje y otra para descifrarlo. La clave para cifrar es la que se conoce como *clave pública*, mientras que la que se utiliza para descifrar es la *clave privada*. Estos criptosistemas surgieron para paliar la debilidad de los criptosistemas simétricos, que es que la clave que cifra y descifra se tiene que compartir, pudiendo esta ser interceptada. La seguridad de estos criptosistemas reside en que no se conozca la clave privada. La información ha sido obtenida de [25].
Un criptosistema asimétrico está formado por:

- $\mathcal{M}$ es el conjunto de los mensajes.
- $\mathcal{C}$ es el conjunto de los criptogramas.
- Una función $P : \mathcal{K}' \to \mathcal{K}$, que nos permitirá generar la clave pública. De manera que para cualquier clave privada $k' \in \mathcal{K}'$ obtenemos la clave pública como $P(k') = k$

Un criptosistema asimétrico viene definido por dos aplicaciones:

$$E : \mathcal{K} \times \mathcal{M} \to \mathcal{C},$$

$$\mathcal{D} : \mathcal{K}' \times \mathcal{C} \to \mathcal{M},$$

y se definen las funciones de cifrado y descifrado como:

$$E_k : \mathcal{M} \to \mathcal{C},$$
$$E_k(m) = E(k, m),$$

$$D_{k'} : \mathcal{C} \to \mathcal{M},$$
$$D_{k'}(c) = D(k', c).$$

Para que un criptosistema asimétrico sea seguro tenemos que garantizar:

- P es una función de dirección única, es decir, que dado un elemento de su imagen no se puede calcular su imagen inversa fácilmente.
- Para la mayoría de los $k \in \mathcal{K}$, la aplicación E_k es de dirección única.

- $\mathcal{D}_{k'}$ se puede calcular en un periodo corto de tiempo si se conoce k' y es imposible o el periodo es muy largo en caso de solo conocerse k.

Capítulo 3

Aritmética modular y cuerpos finitos

En este capítulo presentaré la teoría necesaria de aritmética modular y de cuerpos finitos para comprender adecuadamente las operaciones que se realizan en algoritmos como AES y RSA.

3.1. Aritmética modular

La aritmética modular es una de las piezas fundamentales de la criptografía y es especialmente importante en algoritmos como RSA. A continuación voy presentar unos resultados que serán relevantes a la hora de explicar los distintos algoritmos utilizados.

Definición 3.1. *Se dice que dos elementos a y b están en la misma clase de congruencia si verifican $a \equiv b \mod n$ es decir si ambos tienen el mismo resto al dividir por n o análogamente si $a - b$ es múltiplo de n.*

Definición 3.2. *Una función aritmética f se dice que es multiplicativa si $f(mn) = f(m)(n)$ para cualesquiera enteros positivos que verifiquen $\mathrm{mcd}(n, m) = 1$.*

Definición 3.3. *Se define como función de Moebius a la función*

$$\mu(n) = \begin{cases} 1 & \text{si } n = 1, \\ (-1)^k & \text{si } n \text{ es el producto de } k \text{ primos distintos}, \\ 0 & \text{si } n \text{ tiene algún divisor cuadrado mayor que 1}. \end{cases}$$

Proposición 3.4. *La función $\mu(n)$ es multiplicativa.*

Demostración. Si se cumple $\text{mcm}(m,n) = 1$ entonces tenemos que $m = p_1^{\alpha_1} \dots p_r^{\alpha_r}$ y $n = q_1^{\beta_1} \dots q_s^{\beta_s}$ con $p_i \neq q_j$ para todo $i \in \{1 \dots r\}$ y todo $j \in \{1 \dots s\}$. Si algún α_i o algún β_j es mayor que uno, tendríamos m o n tienen algún divisor cuadrado que también lo será de mn. Luego en este caso tendríamos que $\mu(mn) = \mu(m)\mu(n) = 0$.
Si $\alpha_1 = \dots = \alpha_r = \beta_1 = \dots = \beta_s = 1$ entonces tenemos que $\mu(m) = \mu(p_1, \dots, p_r) = (-1)^r$, $\mu(n) = \mu(q_1, \dots, q_s) = (-1)^s$ y por tanto $\mu(p_1 \dots p_r q_1 \dots q_s) = (-1)^{r+s}$. □

Proposición 3.5.

$$\mu(n) = \begin{cases} 1 & si\ n = 1, \\ 0 & si\ n > 1. \end{cases}$$

Demostración. Como hemos visto en la proposición anterior, $\mu(n)$ es multiplicativa. Por lo tanto tenemos que la función $g(n) = \sum_{d|n} \mu(d)$ también lo es. Si $n = \prod_i q_i^{\alpha_i}$ entonces $g(n) = \prod_i g(q_i^{\alpha_i})$. Pero si q_i es primo, entonces se cumple que $g(q_i^{\alpha_i}) = \sum_{d|q_i} \mu(d) = \mu(1) + \mu(q_i) + \mu(q_i^2) + \dots$
$\dots + \mu(q_i^{\alpha_i}) = 1 - 1 + 0 + \dots + 0 = 0$. □

Definición 3.6. *Se conoce a la función $\phi(n)$ como la función de Euler. Esta es definida como $\phi(n) = |\mathbb{Z}_n^*|$ y calcula el número de enteros positivos que son menores que n y son primos con este.*

Proposición 3.7. *La función $\phi(n)$ es multiplicativa.*

Demostración. Por la proposición anterior tenemos que la función $\phi(n)$ se puede escribir como

$$\phi(n) = \sum_{k \leq n} \sum_{d|(k,n)} \mu(d) = \sum_{d|n} \mu(d) \frac{n}{d} = n \sum_{d|n} \frac{\mu(d)}{d}.$$

Como $\mu(d)$ es multiplicativa, tenemos que $\frac{\mu(d)}{d}$ también lo es. Luego $\frac{\phi(n)}{n}$ es multiplicativa y por tanto $\phi(n)$ también. □

Corolario 3.8.

$$\phi(n) = n \prod_{p_i|n} \left(1 - \frac{1}{p_i}\right).$$

Lema 3.9. *(Gauss)* $\sum_{d|n} \phi(d) = n$

Demostración. La función $f(n) = \sum_{d|n} \phi(n)$ es multiplicativa. Por lo tanto solo nos quedaría ver que $f(p^m) = p^m$ para todo primo p y para todo entero

positivo m.
Para ello hacemos

$$f(p^m) = \sum_{d|p^m} \phi(d) = \phi(1) + \phi(p) + \phi(p^2) + \cdots + \phi(p^m) =$$
$$= 1 + (p-1) + (p^2 - p) + \cdots + (p^m - p^{m-1}) = p^m.$$

□

Teorema 3.10. *(Teorema de Euler) Sean a,n* $\in \mathbb{Z}$ *primos relativos entre sí, entonces* $a^{\phi(n)} \equiv 1 \mod n$.

Demostración. Sea $n \in \mathbb{Z}^+$ que verifica que $\text{mcd}(a, n) = 1$ y definimos S como el conjunto de las unidades modulo n, $S = \{u_1, u_2, \ldots, u_{\phi(n)}\}$ donde $1 \leq u_i \leq n-1$, $\text{mcd}(u_i, n) = 1$ y $u_i \neq u_j \ \forall i, j \in \{1, \ldots, \phi(n)\}$ con $i \neq j$. Multiplicando los elementos de S por a obtenemos

$$aS = \{au_1, au_2, \ldots, au_{\phi(n)}\}$$

Como $\text{mcd}(a, n) = 1$ entonces $a \mod n$ es una unidad y por tanto aS será el conjunto de las unidades módulo n. Y dado que los elementos de S y los de aS coinciden módulo n, el producto de estos será el mismo módulo n por lo que obtenemos

$$u_1 u_2 \ldots u_{\phi(n)} \equiv (au_1)(au_2) \ldots (au_{\phi(n)}) \mod n.$$

Sacando como factor común a tenemos

$$u_1 u_2 \ldots u_{\phi(n)} \equiv a^{\phi(n)} u_1 u_2 \ldots u_{\phi(n)} \mod n.$$

□

Teorema 3.11. *(Teorema pequeño de Fermat) Sea* $a \in \mathbb{Z}$ *y* p *un número primo tal que* $\text{mcd}(a, p) = 1$. *Entonces*

$$a^{p-1} \equiv 1 \mod p.$$

Cabe a destacar que el Teorema de Fermat es un caso particular del Teorema de Euler.

Teorema 3.12. *(Teorema Chino del Resto) Sean* $a_1, a_2 \in \mathbb{Z}$ *y* $p, q \in \mathbb{N}$ *tales que* $\text{mcd}(p, q) = 1$. *Entonces el sistema*

$$x \equiv a_1 \mod p,$$

$$x \equiv a_2 \mod q,$$

tiene solución única módulo $n = pq$. Además, la solución está dada por

$$x \equiv a_1 \cdot q \cdot d_1 + a_2 \cdot p \cdot d_2 \mod n,$$

donde se cumple

$$q \cdot d_1 \equiv 1 \mod p,$$
$$p \cdot d_2 \equiv 1 \mod q.$$

Demostración. Veamos la existencia de la solución. Sea $N = p \cdot q$. Como por hipótesis tenemos que $\text{mcd}(p, q) = 1$, por la identidad Bézout existen enteros d_i, s_i con $i = 1, 2$ tales que $d_1 p + s_1 q = 1$ y $d_2 q + s_2 p = 1$. Tomando módulo p y q tenemos que

$$d_1 \cdot p \equiv 1 \mod q,$$
$$d_2 \cdot q \equiv 1 \mod p.$$

Definiendo

$$x := a_1 \cdot q \cdot d_1 + a_2 \cdot p \cdot d_2,$$

se consigue que x sea la solución del sistema. Además, trabajando módulo p y q obtenemos que $x \equiv a_1 \mod p$ y $x \equiv a_2 \mod q$.
Una vez vista la existencia de la solución quedaría demostrar la unicidad. Para ello supongamos que existen dos números enteros distintos x e y tales que

$$x \equiv a_1 \mod p,$$
$$y \equiv a_1 \mod p.$$

Y también

$$x \equiv a_2 \mod q,$$
$$y \equiv a_2 \mod q.$$

Esto implica que $x - y \equiv 0 \mod p$ y $x - y \equiv 0 \mod q$. Como p y q son coprimos, tenemos que $x - y \equiv 0 \mod N$, luego se da que $x \equiv y \mod N$ llegando así a una contradicción. □

Proposición 3.13. *Sea $n = pq$, donde p y q son dos primos distintos. Si $x \equiv 1 \mod \phi(n)$, entonces $a^x \equiv a \mod n$ para todo $a \in \mathbb{Z}$.*

Demostración. Si a es múltiplo de n, entonces se cumple que $a^x \equiv 0 \equiv a \mod n$. Si a y n son coprimos, $\text{mcm}(a, n) = 1$. Entonces tendríamos que $a^x \equiv a \mod n$ por el Teorema de Euler.
Nos quedaría ver ocurre en el caso de a sea múltiplo de p o de q, pero no de ambos. Por simetría supondremos que a es múltiplo de p, pero no de q. En este caso tenemos que $a^x \equiv 0 \equiv a \mod p$ y $a^x \equiv a \mod q$ por el Teorema de Fermat. Como p y q son coprimos entre sí, aplicando el Teorema Chino del Resto se deduce que $a^x \equiv a \mod n$. □

3.2. El cuerpo de Galois $\mathrm{GF}(2^n)$

Tanto para explicar el funcionamiento de las rondas de AES como para desarrollar la teoría de Curvas Elípticas en $\mathrm{GF}(2^n)$ es necesario introducir el cuerpo $\mathrm{GF}(2^n)$, el cual debido a las propiedades que tiene es muy utilizado en criptografía.

Sea $\mathbb{Z}_2[x]$ el conjunto de polinomios con coeficientes en $\mathbb{Z}_2$, es decir, el conjunto de polinomios cuyos coeficientes solo valen 0 ó 1. Así los polinomios pueden ser representados por una cadena de bits. Un ejemplo sería el polinomio $f(x) = x^4 + x^3 + x + 1$ que quedaría representado como 11011. Además si lo sumamos con otro polinomio como puede ser $g(x) = x^2 + x + 1$, tenemos que $f(x) + g(x) = x^4 + x^3 + x^2$, que equivale a hacer la operación XOR entre 11011 y 00111, por lo que a nivel computacional, es muy fácil implementar estas operaciones.

Podemos definir el cuerpo $\mathrm{GF}(2^n)$ como $\mathbb{Z}_2[x]/(a(x))$, con $a(x)$ un polinomio irreducible en $\mathbb{Z}/(a(x))$. Tenemos que la existencia del inverso de cualquier polinomio no nulo está asegurada por el algoritmo Extendido de Euclídes.

Principalmente se trabaja con $\mathrm{GF}(2^n)$ debido a que la implementación de las operaciones de este es más sencilla que la implementación en las que se utilizan otros cuerpos. Esta sencillez de implementación se traduce en unos algoritmos con tiempos de ejecución menores a pesar de tener el mismo orden de complejidad.

A continuación presentaré el cuerpo $\mathrm{GF}(2^8)$, ya que será necesario para entender adecuadamente las operaciones utilizadas en AES. La información ha sido obtenida de [20] y [1].

Tenemos que $\mathrm{GF}(2^8) = \mathbb{F}_{256}$ por lo que por comodidad trabajaremos con este último.

Por definición tenemos que para p número primo y n un entero primitivo se define el cuerpo $\mathbb{F}_{p^n}$ al único cuerpo existente con p^n elementos. En particular para trabajar con $\mathbb{F}_{256}$ tomamos $p = 2$ y $n = 8$.

Para construir $\mathbb{F}_{256}$ necesitamos un polinomio de grado 8, con coeficientes en $\mathbb{Z}_2$ y que sea irreducible. En total hay 30 polinomios con esas características donde algunos de ellos son $x^8 + x^4 + x^3 + x + 1$, $x^8 + x^4 + x^3 + x^2 + 1$, $x^8 + x^5 + x^3 + x + 1$, $x^8 + x^5 + x^3 + x^2 + 1$, $x^8 + x^5 + x^4 + x^3 + 1$, $x^8 + x^5 + x^4 + x^3 + x^2 + x + 1$ y $x^8 + x^6 + x^3 + x^2 + 1$.

Cabe a destacar que cualquiera de los polinomios serviría para definir $\mathbb{F}_{256}$ y además no habría ninguna diferencia en la seguridad en los criptosistemas que lo utilicen. Para AES se tomó el polinomio $x^8 + x^4 + x^3 + x + 1$, por lo que a partir de ahora trabajaremos con $\mathbb{Z}_2[x]_{a(x)}$ siendo $a(x)$ el polinomio $x^8 + x^4 + x^3 + x + 1$.

Los elementos que conformarán este cuerpo serán clases de equivalencia de polinomios de grado menor que 8. Cada elemento podrá ser representado de tres formas distintas además de la forma polinomial, como número binario,

número hexadecimal y número decimal. Por ejemplo el polinomio $x^5 + x + 1$ quedaría representado como 00100011 de manera decimal, 23 en hexadecimal y 35 en decimal.
Al ser $\mathbb{F}_{256}$ un cuerpo tenemos que tiene dos operaciones, la operación suma que representaremos como + y la operación producto que representaremos como $\cdot$.
La operación + equivale a la suma en $\mathbb{Z}_2$ y usando la notación en binario, tendríamos que equivaldría con la operación XOR como ya he mencionado anteriormente. El opuesto para la suma de un elemento equivalente a sí mismo por lo que no habría diferencia entre sumar por un número o por su apuesto, luego tendríamos que la suma es la misma operación que la resta.
La operación $\cdot$ es mucho más compleja ya que en principio habría que realizar la operación en notación polinomial y luego calcular el resto de dividir por $x^8 + x^4 + x^3 + x + 1$. Para calcular el inverso tendríamos que utilizar el algoritmo extendido de Euclídes. Ambos algoritmos tienen una complejidad algorítmica importante, pero se puede reducir. A continuación recordaré unos resultados de cuerpos finitos que nos permitirán obtener unos métodos que reducirán mucho esa complejidad.

Definición 3.14. *Sea $K = \mathbb{F}_q$ un cuerpo finito ($q = p^n$). Un elemento primitivo de K es un elemento α que tiene $q - 1$ potencias distintas.*

Definición 3.15. *Sea $\alpha \in \mathbb{F}^*$ un elemento no nulo del cuerpo F, definimos orden de α como el menor $d \in \mathbb{N}$ que verifica $\alpha^d = 1$.*

Teorema 3.16. *El orden de todo elemento $\alpha \in \mathbb{F}_q^*$ divide a $q - 1$.*

Demostración. Veamos que $\alpha^{q-1} = 1$. Para ello tomamos el producto de todos los elementos no nulos de $\mathbb{F}_q$ y multiplicando cada miembro por α tenemos que

$$\prod_{\beta_i \in \mathbb{F}_q^*} \alpha\beta_i = \alpha^{q-1} \prod_{\beta_i \in \mathbb{F}_q^*} \beta_i$$

ya que hay $q - 1$ elementos en $\mathbb{F}_q^*$ y por tanto $\alpha^{q-1} = 1$.
A continuación llamamos d al orden de α y suponemos que no divide a $q - 1$, entonces tenemos que existe $r \in \mathbb{N}$ tal que $q-1 = cd+r$ y por tanto se cumple que $\alpha^r = \alpha^{q-1-cd} = 1$ llegando a una contradicción con la minimalidad del orden de d. □

Proposición 3.17. *Sea F un cuerpo finito con p^d elementos, donde p es primo y $d \geq 1$, y sea*

$$x^{p^d} - x = m_1(x)m_2(x)\dots m_n(x)$$

la factorización de x^{p^d} en polinomios irreducibles en $\mathbb{Z}_p[x]$. Entonces:

1. *El polinomio mínimo de cada elemento de* $\mathbb{F}$ *es uno de los polinomios* $m_1(x), \ldots, m_n(x)$.

2. *Para cada* i*, el número de elementos de* $\mathbb{F}$ *con polinomio mínimo* $m_i(x)$ *es igal al grado de* $m_i(x)$.

Demostración. Como cada elemento de $\mathbb{F}$ es una de las raíces de $x^{p^d} - x$, entonces cada $m_i(x)$ tiene que tener un número de raíces en $\mathbb{F}$ equivalente a su grado. Como $m_i(x)$ es irreducible, entonces tienen que haber un polinomio mínimo para cada una de esas raíces. □

Teorema 3.18. *Dos cuerpos finitos con el mismo número de elementos son isomorfos.*

Demostración. Sean $\mathbb{F}_1$ y $\mathbb{F}_2$ dos cuerpos con p^d elementos, donde p es primo y $d \geq 1$. Sea α un generador de $\mathbb{F}_1$. Por la proposición anterior tenemos que el polinomio mínimo $m(x)$ de α tiene que ser un polinomio irreducible de $x^{p^d} - x$ en $Z_p[x]$. Además por la misma proposición tenemos que existe al menos un elemento $\beta \in \mathbb{F}_2$ tal que su polinomio mínimo sea $m(x)$. Como β tiene grado d se cumple que β es generador de $\mathbb{F}_2$ y por tanto tenemos que $\mathbb{F}_1$ y $\mathbb{F}_2$ son isomorfos a $\mathbb{Z}_p[x]/m(x)$ y por tanto isomorfos. □

Teorema 3.19. *Todo cuerpo finito tiene un generador. Si* g *es un generador de* $\mathbb{F}_q^*$*, entonces* g^j *es también un generador si y sólo si* $\gcd(j, q-1) = 1$*. En particular, hay* $\phi(q-1)$ *generadores distintos de* $\mathbb{F}_q^*$.

Demostración. Supongamos $\alpha \in \mathbb{F}_q^*$ con orden d. Veamos primero α^j tiene orden d si y solamente si $\gcd(j, d) = 1$.
Para ver que es condición necesaria tomamos un j que verifique $\gcd(j, d) = d'$ con $d'>1$, tenemos que $a = \frac{d}{d'}$ y $b = \frac{j}{d'}$ son números enteros y además se cumple que $ja = bd'a = bd$ y por tanto $(\alpha^j)^a = (\alpha^d)^b = 1$. Luego se cumple que el orden de α^j es a lo mismo $a<d$.
Veamos que también es condición suficiente.
Para ello tomamos un j que verifique $\gcd(j, d) = 1$ y supongamos que α^j tiene orden $k<d$. Como se cumple que $1 = (\alpha^j)^k = (a^k)^j$ y d es el orden de α tenemos que $(\alpha^k)^d = (a^d)^k = 1^k = 1$.
Por tanto, se cumple que $(\alpha^k)^1 = (\alpha^k)^{\gcd(j,d)} = (\alpha)^{uj+rd} = (\alpha^k)^{uj}(\alpha^k)^{vd} = 1$.
Llegando así a una contradicción, ya que hemos supuesto que α no tiene orden d.
Luego tenemos que el orden de α^j es d.
Como las potencias de $\alpha, (\alpha, \alpha^2, \ldots, \alpha^d = 1)$ cumplen la ecuación $X^d = 1$ y son raíces de las misma, se cumple que no hay más elementos con orden d.
Luego $\mathbb{F}_q^*$ tiene $\phi(d)$ elementos distintos de orden d.
Por último vamos a ver la existencia de un generador g de $\mathbb{F}_q^*$.

Sabemos que la función *phi de Euler* cumple en $q-1$ que $\sum_{d|q-1}\phi(d) = q-1$ y como se cumple que todo elemento de $\mathbb{F}_q^*$ tiene un orden d divisor de $q-1$, necesariamente existen elementos de $\mathbb{F}_q^*$ con orden d para todo $d|q-1$ y en particular se cumple la hipótesis del enunciado. □

Corolario 3.20. *El grupo multiplicativo de todos los elementos no nulos de un cuerpo finito es cíclico.*

Si α es un elemento primitivo de $\mathbb{F}_q$, los $q-1$ elementos de la forma

$$\alpha^0 = 1,\ \ \alpha^1 = \alpha,\ \ldots\ , \alpha^{q-2}$$

serán todos distintos y no nulos. Por tanto serán todos los elementos no nulos de $\mathbb{F}_q$. Además, se verifica que $\alpha^{q-1} = \alpha^0$ por lo que para cualquier $n \in \mathbb{Z}$ se cumple que $\alpha^n = \alpha^{n \ \mathrm{mód}\ q-1}$.
Salvo para $q = 2$, el número de elementos primitivos de $\mathbb{F}_q$ es $\phi(q-1)$. Para el caso $q = 13$ se tiene que $\phi(12) = \phi(2^2 \cdot 3) = 2 \cdot 2 = 4$, luego $\mathbb{Z}_{13}$ tiene 4 elementos primitivos que son 2, 6, 7 y 11. En $\mathbb{F}_{256}$ tenemos que hay $\phi(256) = 128$ elementos primitivos.

Ahora para multiplicar dos elementos pertenecientes a $\mathbb{Z}_{13}$ podemos usar su logaritmo en base 2, el exponente que hay que elevar 2 para obtener el número, sumar los logaritmos y reducirlos base 13 y elevar 2 al resultado. Un ejemplo sería:

$$10 \cdot 12 = 2^{10} \cdot 2^6 = 2^{16} = 2^3 = 8.$$

Para calcular el inverso sería:

$$12^{-1} = (2^6)^{-1} = 2^{12-6} = 2^6 = 12.$$

Esta es la idea que nos permite optimizar las multiplicaciones y los cálculos de inversos en $\mathbb{F}_{256}$. Para ello elegimos un elemento primitivo que nos servirá de generador, en este caso nosotros utilizaremos el más pequeño, que es $[x+1]$ en notación polinomial, 00000011 en binario, 03 en hexadecimal y 3 en binario. Por comodidad trabajaremos en hexadecimal.

Por ejemplo para calcular el resultado de $[x+1]^{125} = (03)^{125}$ lo que hacemos es escribir 125 en hexadecimal que es 7D. A continuación miramos en la tabla, la fila 7 y la columna D que nos da 20 luego tendríamos que $(03)^{125} = 20$. Pasándolo a forma polinomial tenemos que $[x+1]^{125} = x^5$.
Para construir la tabla es mejor usar la forma binaria.

A continuación vamos a construir la tabla inversa de la anterior, esta nos permitirá calcular dado un $z \in \mathbb{F}_{256}$ con $z \neq 0$ el valor de y que verifica $[x+1]^y = z$ que denominaremos $\log_{x+1}(z)$.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	01	03	05	0F	11	33	55	FF	1A	2E	72	96	A1	F8	13	35
1	5F	E1	38	48	D8	73	95	A4	F7	02	06	0A	1E	22	66	AA
2	E5	34	5C	E4	37	59	EB	26	6A	BE	D9	70	90	AB	E6	31
3	53	F5	04	0C	14	3C	44	CC	4F	D1	68	B8	D3	6E	B2	CD
4	4C	D4	67	A9	E0	3B	4D	D7	62	A6	F1	08	18	28	78	88
5	83	9E	B9	D0	6B	BD	DC	7F	81	98	B3	CE	49	DB	76	9A
6	B5	C4	57	F9	10	30	50	F0	0B	1D	27	69	BB	D6	61	A3
7	FE	19	2B	7D	87	92	AD	EC	2F	71	93	AE	E9	20	60	A0
8	FB	16	3A	4E	D2	6D	B7	C2	5D	E7	32	56	FA	15	3F	41
9	C3	5E	E2	3D	47	C9	40	C0	5B	ED	2C	74	9C	BF	DA	75
A	9F	BA	D5	64	AC	EF	2A	7E	82	9D	BC	DF	7A	8E	89	80
B	9B	B6	C1	58	E8	23	65	AF	EA	25	6F	B1	C8	43	C5	54
C	FC	1F	21	63	A5	F4	07	09	1B	2D	77	99	B0	CB	46	CA
D	45	CF	4A	DE	79	8B	86	91	A8	E3	3E	42	C6	51	F3	0E
E	12	36	5A	EE	29	7B	8D	8C	8F	8A	85	94	A7	F2	0D	17
F	39	4B	DD	7C	84	97	A2	FD	1C	24	6C	B4	C7	52	F6	01

Cuadro 3.1: Tabla de los antilogaritmos de $[x + 1]$

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		00	19	01	32	02	1A	C6	4B	C7	1B	68	33	EE	DF	03
1	64	04	E0	0E	34	8D	81	EF	4C	71	08	C8	F8	69	1C	C1
2	7D	C2	1D	B5	F9	B9	27	6A	4D	E4	A6	72	9A	C9	09	78
3	65	2F	8A	05	21	0F	E1	24	12	F0	82	45	35	93	DA	6E
4	96	8F	DB	BD	36	D0	CE	94	13	5C	D2	F1	40	46	83	38
5	66	DD	FD	30	BF	06	8B	62	B3	25	E2	98	22	88	91	10
6	7E	6E	48	C3	A3	B6	1E	42	3A	6B	28	54	FA	85	3D	BA
7	2B	79	0A	15	9B	9F	5E	CA	4E	D4	AC	E5	F3	73	A7	57
8	AF	58	A8	50	F4	EA	D6	74	4F	AE	E9	D5	E7	E6	AD	E8
9	2C	D7	75	7A	EB	16	0B	F5	59	CB	5F	B0	9C	A9	51	A0
A	7F	0C	F6	6F	17	74	49	EC	D8	43	1F	2D	A4	76	7B	B7
B	CC	BB	3E	5A	FB	60	B1	86	3B	52	A1	BC	AA	55	29	9D
C	97	B2	87	90	61	BE	DC	FC	B5	95	CF	CD	37	3F	5B	D1
D	53	39	84	3C	41	A2	6D	47	14	2A	9E	5D	56	F2	D3	AB
E	44	11	92	D9	23	20	2D	89	B4	7C	B8	26	77	99	E3	A5
F	67	48	ED	DE	C5	31	FE	18	0D	63	8C	80	C0	F7	70	07

Cuadro 3.2: Tabla de los logaritmos de $[x + 1]$

A partir de estas dos tablas ya podemos hacer sin mucha dificultad a nivel computacional multiplicaciones y cálculo de inversos. En general para realizar el producto de dos elementos X e Y lo haremos de la siguiente manera:

$$X \cdot Y = A\log((\log_{x+1}(X) + log_{x+1}(Y)) \quad \text{mód } 255).$$

Donde $A\log$ es el antilogaritmo.
Análogamente para calcular el inverso tenemos:

$$X^{-1} = A\log(FF - \log_{x+1}(X)).$$

Por ejemplo para calcular $[x^7 + x^6 + x^4 + x^2 + x] \cdot [x^6 + x^5 + x^4 + x^2 + 1]$ hacemos lo siguiente:

1. Pasamos ambos polinomios a la forma hexadecimal como hemos visto anteriormente, en este caso nos quedaría $D6$ y 75.
2. Nos vamos a la tabla de logaritmos y obtenemos que $\log_{03}(D6) = 6D$ y $\log_{03}(75) = 9F$.
3. Sumamos ambos resultados y lo reducimos módulo 255, parar ello lo pasamos a decimal por comodidad $(109 + 159) \mod 255 = 13$.
4. Calculamos el antilogaritmo de $13 = 0D$ con la tabla y tenemos que vale $F8$ por lo que tenemos que $D6 \cdot 75 = F8$.

Con esto visto ya tendríamos todas las herramientas necesarias para poder describir con mayor profundidad el funcionamiento de los distintos criptosistemas que se utilizan en las aplicaciones de mensajería.

Capítulo 4

Criptografía Simétrica. AES

En este capítulo voy a explicar el cifrado Rijndael AES el cual es un cifrado de bloque simétrico muy utilizado actualmente por aplicaciones como *Telegram*, *WhatsApp* y *FacebookChat* entre otras. Este algoritmo se usa principalmente para cifrar los mensajes ya que es muy eficiente a nivel computacional.

4.1. El algoritmo Rijndael AES

El algoritmo Rijndael llamado así en honor a sus dos autores Joan Daemen y Vicent Rijmen, es un algoritmo de cifrado por bloques que fue adoptado en octubre de 2000 por el NIST (*National Institute for Standards and Technology*) para su empleo en aplicaciones criptográficas no militares en sustitución del algoritmo *DES* después de un proceso de más tres años en los que se buscaba un algoritmo que fuera potente, eficiente y fácil de implementar.
Está diseñado para manejar longitudes de clave y de bloque variables entre los 128 y los 256 bits y aunque estos sean variables, en el estándar adoptado por el Gobierno de Estados Unidos en 2001 [6] establece una longitud fija de bloque de 128 bits y una longitud de clave a escoger entre 128, 192 y 256 bits.
La información para los siguientes apartados de AES la he obtenido de [24] y de [20].

4.2. Estructura de AES

AES es un algoritmo que se basa en aplicar un número determinado de rondas a un valor intermedio denominado *estado* que puede ser representado por una matriz rectangular que posee cuatro filas y N_b columnas. Análogamente, la clave tiene la misma estructura, una matriz de cuatro filas y N_k columnas. El bloque a cifrar o descifrar se traslada directamente byte a byte sobre la matriz de estado de columna en columna $(a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1} \dots)$.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

Cuadro 4.1: Ejemplo de matriz de estado con $N_b = 4$(128 bits).

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Cuadro 4.2: Ejemplo de clave con $N_k = 4$(128 bits).

En otros casos el bloque y la clave pueden ser representados como vectores de registro de 32 bits, donde cada registro esta compuesto por los bytes de la columna correspondiente ordenados en orden descendiente.

Siendo B el bloque que queremos cifrar y S la matriz de estado, el algoritmo AES con n rondas se resume en:

1. Calcular $K_0, K_1, ..., K_n$ subclaves a partar de la clave K.
2. $S \leftarrow B \oplus K_0$.
3. Para $i = 1$ hasta n hacer:

   Aplicar la ronda i-ésima del algoritmo con la subclave K_i.

Como las funciones usadas en cada ronda son invertibles, para descifrar aplicaremos las funciones inversas de las funciones usadas para cifrar en el orden opuesto.

	$N_b = 4$(128 bits)	$N_b = 6$(192 bits)	$N_b = 8$(256 bits)
$N_k = 4$(128 bits)	10	12	14
$N_k = 6$(128 bits)	12	12	14
$N_k = 8$(128 bits)	14	14	14

Cuadro 4.3: Número de rondas en función del tamaño de la clave y bloque.

En el algoritmo AES se define cada ronda como una composición de cuatro funciones invertibles diferentes, formando tres *capas*. Estas funciones tienen un propósito específico.

- **Capa de mezcla lineal:** formada por las funciones *DesplazarFila* y *MezclarColumnas* que permite obtener un alto nivel de difusión a lo largo de varias rondas.
- **Capa no lineal:** formada por la función *ByteSub* y es la aplicación paralela de s-cajas con propiedades óptimas de no linealidad.
- **Capa de adición de clave:** es un simple *o-exclusivo* entre el estado intermedio y la subclave correspondiente a cada ronda.

4.3. Las Rondas de AES

Una vez visto algunas de las propiedades de los cuerpos finitos, ya disponemos de las herramientas necesarias para describir las rondas de AES. Dado que este algoritmo puede aplicarse para longitudes diferentes de bloque y clave, el número de rondas es variable, como se ha visto en 4.3.
Siendo S la matriz de estado y K_i la subclave correspondiente a la ronda i-ésima, cada ronda posee esta estructura:

1. $S \leftarrow ByteSub(S)$,
2. $S \leftarrow DesplazarFila(S)$,
3. $S \leftarrow MezclarColumnas(S)$,
4. $S \leftarrow K_i \oplus S$.

En la última ronda se hacen solo los tres primeros pasos del algoritmo.

ByteSub

La función *ByteSub* es una sustitución no lineal que se aplica a cada byte de la matriz de estado mediante una s-caja 8×8. Se obtiene componiendo dos transformaciones:

1. Cada byte se considera como un elemento del GF(2^8) generado por el polinomio irreducible $m(x) = x^8 + x^4 + x^3 + x + 1$ y es sustituido por su inversa multiplicativa quedando el valor cero inalterado.

2. A continuación se aplica la siguiente transformación afín en GF(2) siendo $x_0, x_1, ..., x_7$ los bits del byte correspondiente e $y_0, y_1, ..., y_7$ los del resultado:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

La función inversa de *ByteSub* es la aplicación inversa de la s-caja de cada byte de la matriz de estado.

DesplazarFila

Esta función desplaza a la izquierda de manera cíclica las filas de la matriz de estado. Cada fila f_i se desplaza un número de posiciones c_i diferente. Mientras que c_0 siempre es igual a cero, el resto de valores vine en función de N_b como se puede ver en 4.4.
La función inversa será el desplazamiento de las filas de la matriz el mismo número de posiciones pero en el sentido contrario.

N_b	c_1	c_2	c_3
4	1	2	3
6	1	2	3
8	1	3	4

Cuadro 4.4: Valores de c_i según el tamaño de bloque N_b

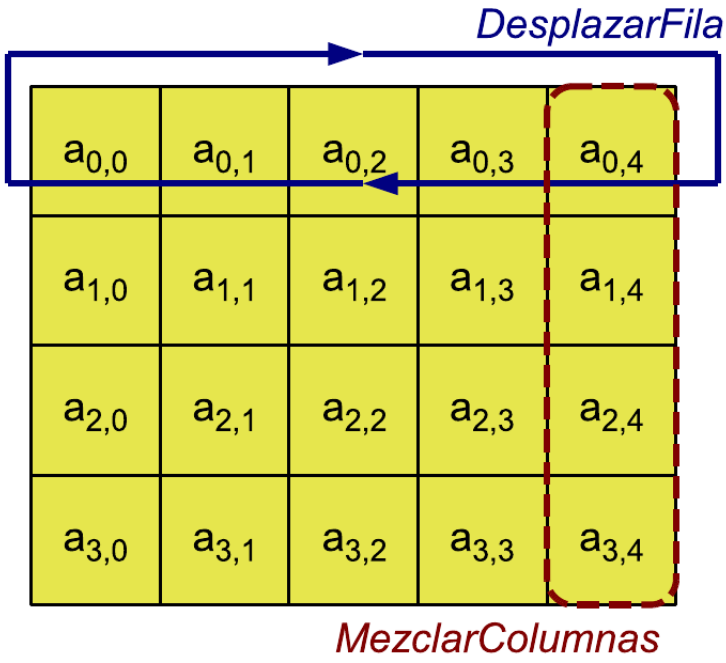


Figura 4.1: Esquema de las funciones *MezclarColumnas* y *DesplazarFila* [24]

MezclarColumnas

Durante la aplicación de esta función cada columna del vector de estado es vista como una matriz 4×1 donde sus coeficientes pertenecen a $\mathbb{F}_{256}$. Aplicar MezclarColumnas a cada estado equivale a multiplicar cada columna por la matriz 4×4.

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$

4.4. Cálculo de las Subclaves

Las subclaves K_i se obtienen de la clave principal K mediante el uso de dos funciones: una de expansión y otra de selección. Siendo n el número de rondas que se van a aplicar, la función de expansión obtiene a partir del valor de K una secuencia de $4(n+1)N_b$ bytes.
La función de selección toma consecutivamente de la secuencia obtenida bloques del mismo tamaño que la matriz de estado y los asigna a cada K_i.

Sea $K(i)$ un vector de bytes de tamaño $4N_k$ conteniendo la clave y sea $W(i)$ un vector de $N_b(n+1)$ registros de 4 bytes, siendo n el número de rondas. La función de expansión tiene dos versiones según el valor de N_k:

- Si $N_k \leq 6$:

Para i desde 0 hasta $N_k - 1$ hacer:
  $W(i) \leftarrow (K(4i), K(4i+1), K(4i+2), K(4i+3))$
Para i desde N_k hasta $N_b(n+1)$ hacer:
  $tmp \leftarrow W(i-1)$
  Si i mód $N_k = 0$
    $tmp \leftarrow Sub(Rot(tmp)) \oplus Rc(i/N_k)$
  $W(i) \leftarrow W(i-N_k) \oplus tmp$

- Si $N_k > 6$:

Para i desde 0 hasta $N_k - 1$ hacer:
  $W(i) \leftarrow (K(4i), K(4i+1), K(4i+2), K(4i+3))$
Para i desde N_k hasta $N_b(n+1)$ hacer:
  $tmp \leftarrow W(i-1)$
  Si i mód $N_k = 0$
    $tmp \leftarrow Sub(Rot(tmp)) \oplus Rc(i/N_k)$
  Si i mód $N_k = 4$
    $tmp \leftarrow Sub(tmp)$
  $W(i) \leftarrow W(i-N_k) \oplus tmp$

La función *Sub* devuelve el resultado de aplicar la s-caja de AES a cada uno de los bytes del registro de cuatro que se le pasa como parámetro, la función *Rot* desplaza a la izquierda los bytes del registro y *Rc(j)* es una constante que se define como:

- $Rc(j) = (R(j), 0, 0, 0)$.
- Cada $R(i)$ es el elemento de $\mathrm{GF}(2^8)$ correspondiente al valor x^{i-1} módulo $x^8 + x^4 + x^3 + x + 1$.

Capítulo 5

Criptografía Asimétrica y Curvas Elípticas. RSA y DH

En esta capítulo voy a explicar de los criptosistemas asimétricos usados en las aplicaciones de mensajería hoy en día. Primero describiré el funcionamiento del cifrado RSA y la firma digital usando este. Después explicaré el problema del Logaritmo Discreto y el intercambio de claves *Diffie-Helmman*. Y por último concluiré haciendo una introducción a la teoría de Curvas Elípticas, el problema del Logaritmo Discreto en estas y el intercambio de claves *Diffie-Hellman* usando estas.

5.1. Criptosistema de Rivest-Shamir-Adleman, RSA

RSA es llamado así en honor a sus creadores Ron Rivest, Adi Shamir y Loenard Adleman. Fue desarrollado en 1977. Cabe a destacar que en 1973 se desarrolló en secreto un criptosistema similar por Clifford Cocks para la *Government Communications Headquarters*, que es la agencia de inteligencia de señales británica, y fue desclasificado en 1997[5].
Una vez que tenemos herramientas necesarias vamos a describir el funcionamiento de RSA. El contenido de esta sección se basa en [25].

5.1.1. Funcionamiento de RSA

El usuario elige dos números primos distintos p y q de buen tamaño ya que mientras más grandes sean más seguro será el cifrado. Se calcula $n = pq$

y por tanto tenemos que $\phi(n) = (p-1)(q-1)$. A continuación se elige un elemento c coprimo con $\phi(n)$ y se calcula el inverso $d = c^{-1} \mod \phi(n)$. La clave pública será $k = (n, c)$ y la clave privada $k' = (n, d)$.
En un principio se consideraba que un tamaño de n de 1024 bits era lo suficientemente grande para que fuera seguro, pero en 2003 Tromer y Shamir mostraron que es posible factorizar números de 1024 bits [29] por lo que en la actualidad se considera 2048 bits como un tamaño seguro.
El conjunto de los mensajes sin cifrar es $\mathcal{M}$, el de los mensajes cifrados será $\mathcal{C}$ y se verifica que $\mathcal{M} = \mathcal{C} = \mathbb{Z}_n$. Las funciones de cifrado y descifrado son respectivamente:

$$\begin{aligned} E_k : \mathcal{M} &\to \mathcal{C}, \\ a &\to a^c, \end{aligned}$$

$$\begin{aligned} D_{k'} : \mathcal{C} &\to \mathcal{M}, \\ a &\to a^d. \end{aligned}$$

Como hemos visto anteriormente, el tamaño del mensaje puede llegar a ser un problema ya que aumenta el tiempo de encriptación y desencriptación. Para solucionar esto se puede fragmentar el mensaje y usar métodos de operación en bloques. Lo más habitual es usar funciones resumen, que de hecho, es el método que se sigue en las aplicaciones de mensajería como veremos en el siguiente capítulo.
Además RSA puede ser vulnerable en función de los números primos que se elijan y el tamaño de estos. Por ejemplo para evitar **el ataque por módulo común**, es recomendable utilizar distinto módulo n para cada clave. Para evitar **el ataque por exponente pequeño** es recomendable utilizar unos números primos p y q grandes. Ya que si no se usan, se puede utilizar el Teorema Chino del Resto para factorizar n y obtenerlos. Para evitar **el ataque por primos muy próximos** se recomienda usar números primos que estén relativamente alejados entre sí.

5.1.2. Firma digital RSA

La firma digital con RSA, es una herramienta muy utilizada en las aplicaciones de mensajería para garantizar el no repudio de los mensajes. Dados dos interlocutores A y B cada uno con sus claves públicas:

- para A tenemos n_A, d_A y e_A,
- para B tenemos n_B, d_B y e_B.

Para que B sepa que un mensaje m ha sido enviado por A se siguen los siguientes pasos.

1. A cifra el mensaje m usando su clave secreta:

$$S = D_A(m) = m^{d_A} \mod n_A.$$

2. A continuación encripta el mensaje firmado con la clave pública de B:

$$C_B(S) = S^{e_B} \mod n_B.$$

y se lo envía a B.

3. B recibe $C_B(S) = S^{e_B} \mod n_B$ y lo desencripta:

$$D_B(S^{e_B}) = S \mod n_B.$$

4. Una vez desencriptado la primera parte, B desencripta S con la clave pública de A:

$$C_A(S) = C_A(D_A(m)) = (m^{d_A})^{e_A} = m^{d_A e_B} = m^{1+k\phi(n_A)} \equiv m \mod n_A.$$

Una vez hecho esto, B podría afirmar casi con total seguridad que el mensaje ha sido enviado por A garantizando el no repudio del mensaje.
Sin embargo este método tiene un inconveniente y es que para documentos muy largos, el proceso para firmar y verificar es muy lento. Para solucionarlo se utiliza una función hash o resumen de manera que en lugar de firmar el mensaje entero, se firma un resumen de este. La firma en este caso quedaría $fir(m) = h(m)^{d_A} \mod n$ y la comprobación sería $h(m) = fir(m)^{d_A} \mod n$ donde h será una función hash o resumen de las que hablaré al final del capítulo.

5.2. El Problema del Logaritmo Discreto. Diffie-Hellman

El intercambio de claves *Diffie-Hellman* es un método basado en el Problema del Logaritmo Discreto muy utilizado en las aplicaciones de mensajería al iniciar una conexión. La información de este apartado sobre el logaritmo discreto ha sido obtenida de [25] y la de *Diffie-Hellman* ha sido obtenida de [24].
El Problema del Logaritmo Discreto es definido de la siguiente forma:

Definición 5.1. *Sea S un semigrupo finito. El Problema del Logaritmo Discreto en el semigrupo S es el de resolver una ecuación del tipo*

$$a^x = b \ (x \in \mathbb{N}),$$

donde a y b son dos elementos dados de S.

La complejidad del Problema del Logaritmo Discreto depende en gran medida del semigrupo S que se elija. Dado que si se eligiera como S el grupo aditivo $\mathbb{Z}_n$ la solución se obtendría fácilmente resolviendo una ecuación de congruencias del tipo $aX \equiv b \mod n$ que equivaldría a resolver la ecuación diofántica $aX + nY = b$. Pero si ahora S pasara a ser los semigrupos multiplicativos $\mathbb{Z}_n$ o $\mathbb{F}_q$ o sus grupos de unidades, el problema aumentaría su complejidad de manera significativa.
Tenemos que $a^x = b$ tiene solución si y solamente si b está en el semigrupo cíclico generado por a. Luego si a es un elemento de orden finito de un grupo, se podría suponer en la práctica que S es un grupo cíclico y por ello existiría un isomorfismo con $(\mathbb{Z}_n, +)$. Luego la dificultad del problema no estaría en la estructura del grupo, sino en reconocer los elementos como potencias de enteros.
Se cree que el problema de logaritmo discreto es $\mathbb{NP}$-completo, pero esta conjetura todavía no ha sido demostrada por lo que se considera un problema $\mathbb{NP}$-Intermedio. Estos problemas son llamados así porque no están dentro de los problemas $\mathbb{P}$ ni en los problemas $\mathbb{NP}$-completo [10].
Una vez visto el problema de logaritmo discreto, se explicará el intercambio de claves *Diffie-Hellman*.

5.2.1. Intercambio de claves Diffie-Hellman

Antes de explicar el intercambio de claves *Diffie-Hellman* se introducirá el problema de Diffie-Hellman ya que es la base de este.

Definición 5.2. *(El Problema Diffie-Hellman)*
Dado un número primo p, un número α que sea un generador *de $\mathbb{Z}_{p'}^*$, α^a y α^b, encontrar $\alpha^{ab} \mod p$.*

Intercambio de claves *Diffie-Hellman*

El intercambio de claves *Diffie-Hellman* es un algoritmo asimétrico basado en el problema de *Diffie-Hellman*, empleado para acordar una clave común en un canal inseguro. Los pasos que se siguen son:
Sean A y B dos interlocutores que quieren compartir un valor K. Para ello se calcula un número primo p y un generador α de $\mathbb{Z}_{p'}^*$ con $2 \leq \alpha \leq p-2$. Esta información es pública y conocida por ambos.

1. A escoge un número aleatorio x, comprendido entre 1 y $p-2$ y envía a B el valor
$$\alpha^x \mod p.$$

2. Análogamente B escoge un número aleatorio y, comprendido entre 1 y $p-2$ y envía a A el valor
$$\alpha^y \mod p.$$
3. B recoge α^x y calcula $K = (\alpha^x)^y \mod p$.
4. A recoge α^y y calcula $K = (\alpha^y)^x \mod p$.

Puesto que x e y son conocidos solamente por A y B respectivamente, tenemos que al final ambos acaban conociendo el valor de K.

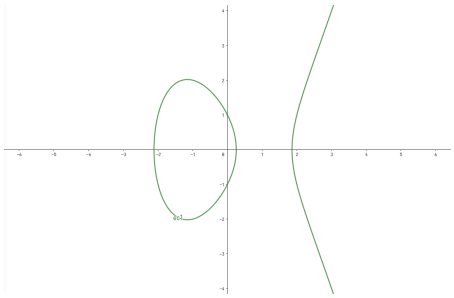
5.3. Curvas Elípticas en Criptografía

A continuación se introducirá la teoría de curvas Elípticas ya que nos permitirá redefinir el intercambio de claves usando estas. Es el método que se usa actualmente en aplicaciones de mensajería que implementan el protocolo *MTProto* y *Signal*.
La criptografía en Curvas Elípticas es considerada como uno de los campos con mayor potencial en la criptografía asimétrica. Esto es debido a sus propiedades que dan lugar a problemas muy complejos análogos a los que presenta la aritmética modular por lo que garantizan más la seguridad. Esto permite que sean utilizadas en algunos algoritmos asimétricos como puede ser el intercambio de claves *Diffie-Hellman* que se verá más adelante. Aunque su estructura algebraica es más compleja que la de la aritmética modular sin embargo, al implementarlas suelen ser más eficientes y además, con claves más cortas alcanzan el mismo nivel de seguridad.
El uso de curvas elípticas en criptografía se presentó por primera vez en 1985 por Neal Koblitz y Víctor Miller de manera independiente.
La información para esta sección se ha obtenido de [24] y de [23].

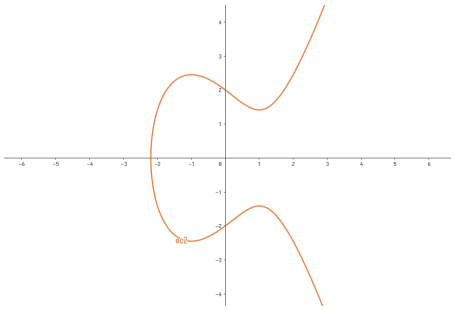
Una curva elíptica E sobre un cuerpo $\mathcal{K}$ es el conjunto de los $(x, y) \in \mathcal{K}^2$ tales que
$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$
junto con un punto $\mathcal{O}$.
Un ejemplo de curvas elípticas en $\mathbb{R}$ son:

Proposición 5.3. *Sea $p = \mathrm{char}(\mathcal{K})$ y sea E una curva elíptica definida sobre $\mathcal{K}$.*

1. *Si $p>3$, la ecuación E puede simplificarse a*
$$y^2 = x^3 + ax + b, \tag{5.1}$$
recibiendo el nombre de forma de Weiertrass.



(a) Curva $y^2 = x^3 - 4x + 1$



(b) Curva $y^2 = x^3 - 3x + 4$

2. *Si $p = 3$, la ecuación de E puede simplificarse a*

$$y^2 = x^3 + a^2 + bx + c. \tag{5.2}$$

3. *Si $p = 2$ la ecuación de E puede simplificarse a*

$$y^2 + xy = x^3 + ax^2 + b \; si \; a_1 \neq 0, \tag{5.3}$$
$$y^2 + ay = x^3 + bx + c \; si \; a_1 = 0, \tag{5.4}$$

también se denominan como forma de Weiertrass.

Las curvas más utilizadas en criptografía son las curvas definidas sobre $\mathbb{Z}_p$ del tipo (5.1) y las curvas sobre $\mathbb{F}_{2^l}$ con un l grande y verificando (5.3). Me voy centrar en explicar estas últimas porque actualmente en las aplicaciones de mensajería son las que se utilizan.

Proposición 5.4. *Una curva $E(x, y)$ sobre $\mathbb{F}_{2^l}$ que satisface* (5.3) *tiene un punto singular si y solo si $b = 0$.*

Demostración. Sea $F(x, y) = y^2 + xy + x^3 + ax^2 + b$. E es singular en $(x_0, y_0) \in E$ si y solo si

$$\frac{\partial F}{\partial x}(x_0, y_0) = \frac{\partial F}{\partial y} = 0.$$

Dado que

$$\frac{\partial F}{\partial x} = y + 3x^2 + 2ax = y + x^2$$

y

$$\frac{\partial F}{\partial y} = 2y + x = x,$$

el único punto donde puede haber una singularidad es $(0, 0)$ que pertenece a la curva si y solo si b=0. □

Lema 5.5. *Sea $E = E(a,b)$ una curva sobre $\mathbb{F}_{2^l}$ definida por la ecuación (5.3). Si $(x_0,y_0),(x_1,y_1) \in E$, entonces $y_1 = y_0$ o $y_1 = x_0 + y_0$.*

Demostración. Como se cumple que

$$y_0^2 + x_0y_0 = x_0^3 + ax_0^2 + b = y_1^2 + x_0y_1,$$

tenemos que

$$(y_1 + y_0)^2 = y_1^2 + y_0^2 = (y_1 + y_0)x_0.$$

Si $y_0 \neq y_1$ tenemos que $y_0 + y_1 \neq 0$ y por tanto $x_0 = y_1 + y_0$, luego $y_1 = y_0$ o $y_1 = y_0 + x_0$. □

Proposición 5.6. *Sea $E(a,b)$ una curva elíptica sobre $\mathbb{F}_{2^l}$ que verifica (5.3). Sean $P = (x_0,y_0)$, $P_1 = (x_1,y_1)$ y $P_2 = (x_2,y_2)$ puntos de $E(a,b)$. Entonces,*

1. *$-P = (x_0, x_0 + y_0)$.*
2. *Si $P_2 = -P_1$, $P_1 + P_2 = \mathcal{O}$.*
3. *Si $P_2 \neq -P_1$, $P_1 + P_2 = P_3$, viene dado por*

$$\begin{aligned} P_3 &= (x_3,y_3) \\ &= (m^2 + m + a + x_1 + x_2, m(x_1 + x_3) + x_3 + y_1), \end{aligned}$$

donde

$$m = \begin{cases} (y_2 + y_1)(x_2 + x_1)^{-1} & si\ x_1 \neq x_2, \\ x_1 + y_1x_1^{-1} & si\ x_1 = x_2 \end{cases}$$

Demostración. La ecuación (5.1) es un caso particular de (5.3) tomando $a_1 = 1$, $a_3 = 0$, $a_2 = a$, $a_4 = 0$ y $a_6 = b$. Luego la aritmética es consecuencia de la aritmética definida para una curva elíptica cualquiera, por el lema visto anteriormente, tenemos que $x_1 = x_2$ y $P_2 \neq -P_1$ implica $P_2 = P_1$ y por tanto $P_1 + P_2 = 2P_1$. □

5.3.1. Uso de las Curvas Elípticas en criptografía

Una vez introducidas las Curvas Elípticas y visto unos resultados necesarios para este apartado, voy a explicar los elementos que se necesitan para poder usar las Curvas Elípticas en criptografía.
Para poder utilizarlas necesitaremos los siguientes parámetros.

- El cuerpo base sobre el que se definirán las curvas, $\mathbb{F}_q$
- Los parámetros $a, b \in \mathbb{F}_q$, que serán los que definan la curva E a partir de las ecuaciones (5.1) y (5.3).

- Un punto base $Q \in E$ cuyo orden es n el cual será un primo grande.
- El cofactor h tal que $|E| = hn$. Las curvas que satisfacen esto y además n es primo y h pequeño se denominan *curvas de orden próximo a primo* y E_n es el *subgrupo de orden primo.*

Lema 5.7. *Sea E una curva elíptica tal que $|E| = hn$ con n primo y $h<n$. Entonces E tiene un único subgrupo E_n de orden n que es cíclico y generado por cualquiera de sus elementos distintos de $\mathcal{O}$.*

Demostración. Por el Teorema de Cassel, $E \cong \mathbb{Z}_{d_1} \times \mathbb{Z}_{d_2}$, con $d_1 \mid d_2$. Como $h<n$ se cumple que $n \mid d_2$ pero $n \nmid d_1$. Luego E_n se corresponde con el subgrupo $\{0\} \times \langle \frac{d_2}{n} \rangle \leq \mathbb{Z}_{d_1} \times \mathbb{Z}_{d_2}$. □

Una vez visto esto quedaría ver la selección de la curva a utilizar, ya que como se verá a continuación, no sirve cualquiera. Esto es debido a que necesitamos que el problema del logaritmo discreto sea difícil, es decir, que sea computacionalmente muy costoso de romper.
Una vez seleccionada la curva tendríamos que seleccionar los puntos de la curva y la selección del punto base.
Las familias de curvas que tenemos que evitar utilizar son las siguientes.

- *Curvas supersingulares*, como pueden ser las que tienen la forma (5.4). En estas curvas se cumple que $E_n \cong \mathbb{F}_{q^l}^*$ con l pequeño. Esto fue demostrado por Menezes-Okamoto-Vanstone empleando el llamado *par de Weil*. Aunque en realidad basta con ver que $n \nmid q^l - 1$ para valores pequeños de l.
- Curvas sobre $\mathbb{F}_p$ tales que $|E| = p$. En este caso, Semaev, Smart y Satoh-Araki construyen un isomorfismo $E \cong \mathbb{F}_p$ mediante un algoritmo en tiempo polinomial, reduciendo el problema del logaritmo discreto al uso del algoritmo de Euclídes extendido en $\mathbb{F}_p$.

Teniendo en cuenta esto, la curva debe elegirse mediante una búsqueda aleatoria para evitar centrarse en familias que en un futuro pudieran ser comprobadas como inseguras. Por lo que el proceso sería elegir los parámetros $a, b \in \mathbb{F}_q$. A continuación se calcula $|E(a,b)|$ y se observa si $|E(a,b)| = hn$ para h pequeño con n primo. Por último se comprueba que no es vulnerable a los ataques anteriores.
La parte más compleja del procedimiento es calcular el orden de la curva. Pero para ello podemos usar el Termoa de Hasse el cual nos da una cota de este.

Teorema 5.8. *(Hasse). Sea E una curva elíptica sobre $\mathbb{F}_q$ dada por* (5.1) *y sea $t = q + 1 - |E|$. Entonces*

$$|t| \leq 2\sqrt{q}.$$

Gracias a un algoritmo desarrollado por Schoof-Elkies-Atkin para cuerpos primos y a Satoh para cuerpos binarios permite calcular dicho orden en tiempo polinomial.
Para concluir tenemos que existen suficientes curvas con orden próximo a primo para que una búsqueda aleatoria sea efectiva en la práctica. Si el cuerpo base es primo, también existen muchas curvas de orden primo. En característica 2 tenemos que el orden es par.

Para calcular los puntos de la curva tenemos que, para una curva elíptica $E = E(a, b)$ definida sobre $\mathbb{F}_q$ el morfismo

$$\pi : E\backslash\{\mathcal{O}\} \to \mathbb{F}_q,\ (x, y) \mapsto x,$$

no es sobreyectivo. Esto hace que no todos los elementos de $\mathbb{F}_q$ son primera coordenada de un punto de la curva.

Caso $\mathbb{F}_p$
La curva es del tipo (5.1), por lo que debemos buscar valores $x_0 \in \mathbb{F}_p$ que verifiquen $x_0^3 + ax_0 + b$ es un cuadrado perfecto. Por lo que necesitamos decidir si $\beta \in \mathbb{F}_p$ es residuo cuadrático y, en caso de serlo, calcular sus raíces. Para esto nos apoyamos del siguiente lema.

Lema 5.9. *(Criterio de Euler). $\beta \in \mathbb{F}_p$ es residuo cuadrático si y solo si $\beta^{\frac{p-1}{2}} \equiv 1 \mod p$.*

Caso $\mathbb{F}_{2^l}$
En este caso la curva elíptica $E = E(a, b)$ viene dada por la ecuación (5.3). Si $(x_0, y_0) \in E$, y_0 es solución de la ecuación cuadrática

$$y^2 + x_0 y = x_0^3 + ax_0^2 + b.$$

Y por último nos quedaría seleccionar el punto base.
Para ello seleccionamos aleatoriamente $P \in E$ donde $E = E(a, b)$ es una curva elíptica tal que $|E| = hn$ con n primo y h pequeño. Una vez seleccionado calculamos $Q = hP$ y comprobamos si $Q \neq \mathcal{O}$. Como n es primo y $nQ = \mathcal{O}$, Q será un generador de E_n. Si $Q = \mathcal{O}$ tomamos un nuevo P y repetimos la operación.
Con esto ya tenemos todo lo necesario para explicar el Problema del Logaritmo Discreto usando Curvas Elípticas.

5.4. El Problema del Logaritmo Discreto usando Curvas Elípticas. *Diffie-Hellman*

En esta sección voy a describir el análogo del Problema del Logaritmo Discreto en Curvas Elípticas y como resultado un análogo del intercambio de claves *Diffie-Hellman*.

5.4.1. El Problema del Logaritmo Discreto en Curvas Elípticas

Para todo punto p definido en una curva elíptica, se define $\langle p \rangle$ al conjunto $\{\mathcal{O}, p, 2p, ...\}$. En $E(\mathrm{GF}(n))$ y $E(\mathrm{GF}(2^n))$ los conjutos como los que se han definido, tienen que ser finitos ya que los puntos de las curvas son finitos. Luego para todo punto $q \in \langle p \rangle$ tiene que existir un número $k \in \mathbb{Z}$ que verifique que $kp = q$.
Por lo tanto, el problema del logaritmo discreto en curvas elípticas consiste en hallar dicho número k a partir de p y q.

5.4.2. Intercambio de claves *Diffie-Hellman* en curvas elípticas

Una vez visto el Problema del Logaritmo Discreto en Curvas Elípticas, voy a explicar el intercambio de claves *Diffie-Hellman* usando Curvas Elípticas. Para ello se explicaré previamente la conjetura *Diffie-Hellman* en Curvas Elípticas. La información de este apartado la he obtenido de [23].
Fijamos una curva elíptica $E = E(a, b)$ tal que $|E| = hn$ con n primo y h pequeño como he explicado en la sección anterior. Esta curva puede estar definida sobre $\mathbb{F}_p$ o sobre $\mathbb{F}_{2^l}$. También se fija q un elemento de orden n, en función del cuerpo que elijamos lo fijamos de una forma u otra como ya hemos visto.

Definición 5.10. *(Conjetura Diffie-Hellman). Conocidos $p_a = aq$ y $p_b = bq$ para ciertos $1 \leq a, b \leq n$, calcular abq es equivalente a nivel computacional a calcular $a = \log_q(p_a)$ o $b = \log_q(p_b)$.*

El protocolo de intercambio de claves queda como sigue.
Dadas dos personas A y B que quieren realizar un intercambio de claves.

- A y B se ponen de acuerdo en la curva elíptica E y el punto $q \in E$.
- A elige aleatoriamente un número $a \in \{2, \ldots, n-1\}$ y le envía a B $p_a = aq$.

- B elige aleatoriamente un número $b \in \{2, \ldots, n-1\}$ y le envía a A $p_b = bq$.
- A calcula $a(p_b)$.
- B calcula $b(p_a)$.
- La clave compartida es $(ab)q = a(p_b) = b(p_a)$.

Capítulo 6

Funciones Hash

En este capítulo voy a describir que son las funciones hash, como construirlas usando la construcción *Merkle-Damgård* y por último explicaré las funciones hash más utilizadas en las aplicaciones de mensajería en la actualidad.

6.1. ¿Que son las funciones hash?

Una función resumen o función hash es una función que se puede calcular mediante un algoritmo en el cual se transforma un conjunto arbitrario de datos en una nueva serie de caracteres con una longitud fija independiente del tamaño de los datos de entrada. Estas funciones son muy utilizadas en criptografía ya que como hemos visto, en el caso de RSA el tamaño del bloque puede suponer un problema debido a que aumenta considerablemente el tiempo de ejecución. Para ello se utilizan funciones hash, ya que mediante el uso de estas, se consigue reducir el tamaño del mensaje sin perder información.
La información para esta sección la he obtenido [16].

Las propiedades esperadas de una función hash son:

- Se tiene que poder utilizar en contenido digital de cualquier tamaño y formato.
- Independientemente del tamaño de la entrada y del tipo, se produce una salida numérica de tamaño fijo.
- Para el mismo conjunto de datos de entrada, el resultado siempre es el mismo.

- Reconstruir el mensaje original a partir del generado tiene que ser muy complejo, idealmente imposible.
- Una variación mínima del mensaje original tiene que producir un hash totalmente distinto, esta propiedad se denomina *difusión*.
- Dado un mensaje, tiene que ser muy difícil encontrar otro mensaje con la misma imagen que este *colisión débil*.
- Tiene que ser muy costoso encontrar dos mensajes que tengan la misma imagen, esta propiedad es denominada *colisión fuerte*.
- Dado un posible valor del espacio imagen, tiene que ser igual de probable que salga este u otro cualquiera. Es decir todos los valores tienen la misma probabilidad de salir.

Visto esto, en general, una función hash funciona de la siguiente forma:

1. El mensaje de entrada se divide en bloques.
2. Una fórmula calcula el hash, un valor con un tamaño fijo, para el primer bloque.
3. Se calcula el hash del siguiente bloque y se suma con el hash calculado previamente.
4. Se repite de manera análoga con el resto de bloques hasta que se recorren todos.

Las hash que explicaré serán: *SHA-0* y *SHA-1* que son las funciones antecesoras de la función *SHA-256* que es la que se utiliza mayoritariamente en las funciones de mensajería en la actualidad. Algunas también pueden utilizar SHA-1

6.2. La construcción *Merkle-Damgård*

Antes de desarrollar las funciones hash mencionadas anteriormente, voy a describir la construcción *Merkle-Damgård*. Este es un método para construir funciones hash que sean resistentes a colisiones a partir de funciones de compresión unidireccionales. La información para este apartado ha sido obtenida de [30].
Este método fue descubierto en 1989 por Merkle y Damgård de manera independiente. Es muy importante ya que gracias a este, se desarrollaron las funciones hash *MD4, MD5, SHA-0, SHA-1 y SHA-2* entre otras.

Una función de compresión es una función que toma una entrada de longitud fija y produce un resumen de longitud fija. Esta recibe dos entradas: una variable cadena y un bloque del mensaje.

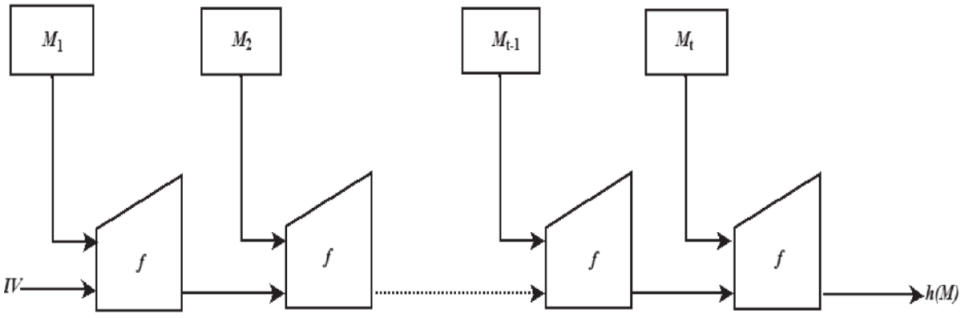


Figura 6.1: Construcción de Merkle-Damgård [30].

Sea $f : \{0,1\}^b \times \{0,1\}^n \to \{0,1\}^n$ una función de compresión que toma un bloque de mensajes de b bits y un valor de encadenamiento de n bits. Sea $h : \{0,1\}^* \to \{0,1\}^n$ una construcción *MD* construida iterando la función de compresión f para procesar un mensaje de longitud arbitraria. Para que esto sea posible lo que se hace es rellenar cualquier mensaje para que la longitud de este sea múltiplo de la longitud de bloque b de f.
El relleno se realiza añadiendo después del último bit del último bloque del mensaje un único bit 1 seguido del resto de bits necesarios a 0. Por último, para terminar de rellenar el mensaje, se añade la codificación binaria de la longitud del mensaje.
Una vez rellenado el mensaje, la entrada se divide en t bloques, cada uno de longitud b. La función hash resultante puede describirse de la siguiente manera.

$$H_0 = IV,$$
$$H_i = f(H_{i-1}, M_i)\ i = 1 \ldots t,$$
$$h(M) = H_t.$$

Este proceso se continua recursivamente, actualizando la variable cadena a partir de pasar cada bloque del mensaje por la función de compresión. La salida será la variable cadena. En 6.2 se puede ver un resumen del proceso.

6.3. SHA-0

SHA-0 es una función hash que apareció publicada en el Federal Information Processing Standard (FIPS-180) por el NIST en 1993 [28]. Está

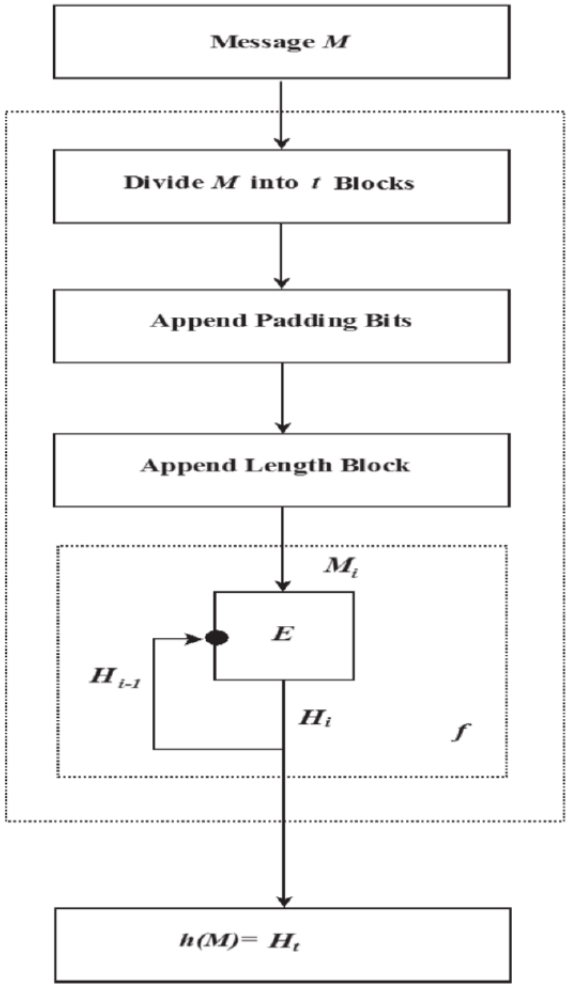


Figura 6.2: Esquema de los pasos seguidos en la construcción de Merkle-Damgård [30].

basada en *MD4* y *MD5*. El algoritmo transforma un mensaje de cualquier tamaño hasta 2^{64} bits y los transforma en un resumen de 160 bits.
El funcionamiento de SHA-0 es el siguiente [17]:

1. Al igual que en MD5, el mensaje se rellena con un único bit '1' seguido de 0-511 bits '0'. A continuación se añade una representación de 64 bits de la longitud del mensaje donde el número de ceros es elegido para asegurar que la longitud total del mensaje es un múltiplo de 512 bits. El mensaje se divide en bloques de 512 bits: $M_1, ..., M_n$.

2. Para la primera iteración se utiliza un buffer predefinido:

   $h_0 = (67452301_x, EFCDAB89_x, 98BADCFE_x, 10325476_x, C3D2E1F0_x)$.

3. Cada bloque M_j es pasado por la función de compresión junto con el valor actual de h_{j-1}, la salida es el nuevo valor de h_j, la operación se puede resumir en

$$h_j = compresión(M_j, h_{j-1}).$$

4. h_n es la salida de la función hash.

Los pasos seguidos en la función de compresión son:

1. Se divide el bloque M_j de 512 bits en bloques 16 bloques de 32 bits $W_0, W_1, ..., W_{15}$.

2. Se expanden los 16 bloques de 32 bits en 80 bloques a partir de la siguiente ecuación en recurrencias:

$$W_i = W_{i-3} \oplus W_{i-8} \oplus W_{i-14} \oplus W_{i-16},\ i = 16, ..., 79.$$

   Esta expansión se nota como exp(.).

3. Divide h_{j-1} en 5 registros A, B, C, D y E como:

$$h_{j-1} = (A_0, B_0, C_0, D_0, E_0).$$

4. Para $i = 0, ..., 79$ hacemos:

$$A_{i+1} = (A_i \lll 5) + f_i(B_i, C_i, D_i) + E_i + K_i) \quad \text{mód } 2^{32},$$

$$B_{i+1} = A_i,\ C_{i+1} = (B_i \lll 30),\ D_{i+1} = C_i,\ E_{i+1} = D_i.$$

   Donde las funciones y las constantes están definidas en la tabla 6.1.

5. La salida de la función sería:

$$h_n = (A_0 + A_{80}, B_0 + B_{80}, C_0 + C_{80}, D_0 + D_{80}, E_0 + E_{80}).$$

Rondas	$f_i(B, C, D)$	K_i
$0 \leq i \leq 19$	$BC \vee BD$	$5AD9EBA1_x$
$20 \leq i \leq 39$	$B \oplus C \oplus D$	$6ED9EBA1_x$
$40 \leq i \leq 59$	$BC \vee BD \vee CD$	$8F1BBCDC_x$
$60 \leq i \leq 79$	$B \oplus C \oplus D$	$CA62C1D6_x$

Cuadro 6.1: Funciones y constantes usadas en la función de compresión de SHA-0 [17].

6.4. SHA-1

La función SHA-1 es una función hash diseñada en 1995 por la *National Security Agency* (NSA) dado que que se encontraron varias colisiones y vulnerabilidades en la función SHA-0 [28].
Su funcionamiento es muy similar al de la función SHA-0 variando en las funciones y variables usadas en las distintas rondas de la función de compresión. En la tabla 6.2 se pueden ver los nuevos valores utilizados.

Rondas	$f_i(B,C,D)$	K_i
$0 \leq i \leq 19$	$(B \wedge C) \oplus (\overline{B} \wedge D)$	$5A827999_x$
$20 \leq i \leq 39$	$B \oplus C \oplus D$	$6ED6EBA1_x$
$40 \leq i \leq 59$	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	$8FABBCDC_x$
$60 \leq i \leq 79$	$B \oplus C \oplus D$	$CA62C1D6_x$

Cuadro 6.2: Funciones y constantes usadas en la función de compresión de SHA-1 [22].

En imagen 6.3 se puede observar un esquema del proceso para obtener un el hash seguido por las funciones SHA-0 y SHA-1 donde **F** será la función de compresión.

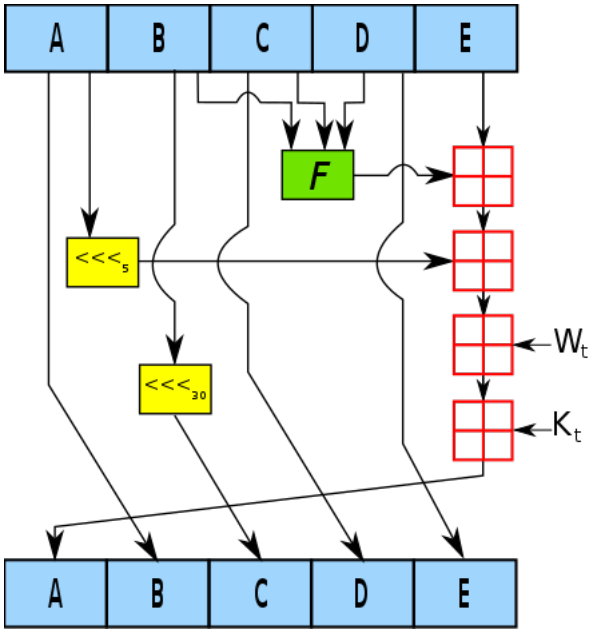


Figura 6.3: Esquema de los pasos seguidos en SHA-0 y SHA-1 [12].

6.5. SHA-256

La función SHA-256 pertenece a la familia SHA-2 que es un conjunto de funciones hash diseñadas por la NSA en 2001 [28]. Esta familia está compuesta por las funciones SHA-224, SHA-256, SHA-384 y SHA-512 donde el número del final indica el tamaño de bloque en el que se dividirá el mensaje. Nos centraremos en la función SHA-256 que como he comentado anteriormente es la que se utiliza en las aplicaciones de mensajería actualmente.
El funcionamiento de las función es el siguiente[14]:

1. Al igual que en SHA-0 y SHA-1 se rellena el mensaje de la misma manera y se fragmenta en bloques de 512 bits: $M_1, ..., M_n$.

2. Para la primera iteración se utiliza un buffer predefinido:

$$h_0 = (H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8),$$

   donde:
   $H_1 = 6A09E776$,
   $H_2 = BB67AE85$,
   $H_3 = 3C6EF372$,
   $H_4 = A54FF53A$,
   $H_5 = 510E527F$,
   $H_6 = 9B05688C$,
   $H_7 = 1F83D9AB$,
   $H_8 = 5BE0CD19$.

3. Cada bloque M_j es pasado por la función de compresión junto con el valor actual de h_{j-1}, la salida es el nuevo valor de h_j, la operación se puede resumir en:

$$h_j = compresión(M_j, h_{j-1}).$$

4. h_n es la salida de la función hash.

Los pasos seguidos en la función de compresión son:

1. Se divide el bloque M_j de 512 bits en bloques 16 bloques de 32 bits $W_0, W_1, ..., W_{15}$.

2. Se expanden los 16 bloques de 32 bits en 63 bloques a partir de la siguiente ecuación en recurrencias:

$$W_i = \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-16}),\ i \in \{16...63\}.$$

3. Divide h_{j-1} en *A, B, C, D, E, F, G* y *H* como:

$$h_{j-1} = (A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0).$$

4. Para $i = 0, ..., 63$ hacemos:

$$A_{i+1} = H_i + \Sigma_1(E_i) + Ch(E_i, F_i, G_i) + K_j + W_j + \Sigma_0(A_i) + Maj(A_i, B_i, C_i),$$

$$B_{i+1} = A_i,\ C_{i+1} = B_i,\ D_{i+1} = C_i,\ F_{i+1} = E_i\ G_{i+1} = F_i,\ H_{i+1} = G_i,$$

$$E_{i+1} = D_i + H_i + \Sigma_1(E_i) + Ch(E_i, F_i, G_i) + K_j + W_j.$$

Donde las funciones y las constantes están definidas en la tabla 6.1.

5. La salida de la función es:

$$h_j = (A_0 + A_{63}, B_0 + B_{63}, C_0 + C_{63}, D_0 + D_{63}, E_0 + E_{63}, F_0 + F_{63}, G_0 + G_{63}, H_0 + H_{63}).$$

Donde tenemos que:

$$Ch(x, y, z) = (x \wedge y) \oplus (\overline{x} \wedge z),$$

$$Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z),$$

$$\Sigma_0(x) = (x \ggg 2) \oplus (x \ggg 13) \oplus (x \ggg 22),$$

$$\Sigma_1(x) = (x \ggg 6) \oplus (x \ggg 11) \oplus (x \ggg 25),$$

$$\sigma_0(x) = (x \ggg 7) \oplus (x \ggg 18) \oplus (x \lll 3),$$

$$\sigma_1(x) = (x \ggg 17) \oplus (x \ggg 19) \oplus (x \lll 10).$$

En la imagen 6.4 podemos ver un esquema de los pasos seguidos en las funciones SHA-2.

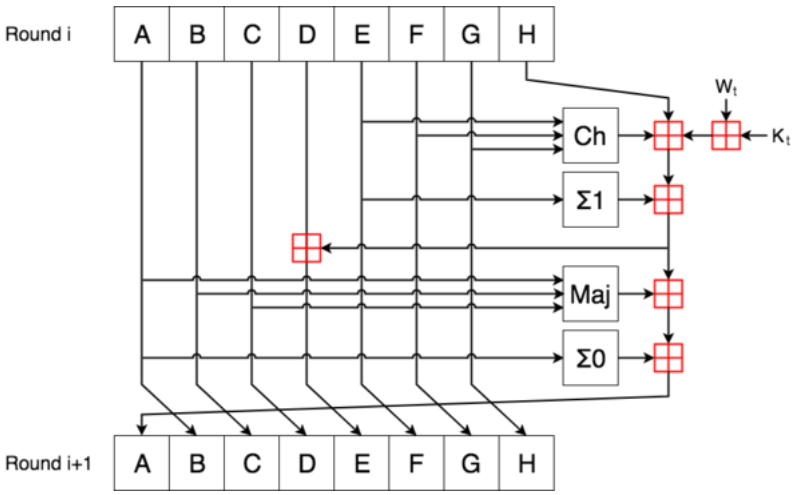


Figura 6.4: Esquema de los pasos seguidos en las funciones de la familia SHA-2 [13].

Con este capítulo concluye la parte de descripción de criptosistemas y herramientas utilizadas por las aplicaciones de mensajería. En el siguiente capítulo voy a describir como utilizan estas herramientas las aplicaciones de mensajería más populares para garantizar la seguridad y la privacidad de sus usuarios.

Capítulo 7

Aplicaciones de Mensajería

En este capítulo explicaré como usan las aplicaciones de mensajería los criptosistemas que he desarrollado en los capítulos anteriores. Las aplicaciones que he estudiado son: *Telegram*, *WhatsApp*, *Facebook*, *Signal*, *iMessage* y *Line Messenger*.

7.1. Telegram (MTProto)

MTProto es el protocolo de datos con el Telegram cifra sus mensajes. Fue desarrollado por el matemático Nikolái Dúrov y financiado por Pável Dúrov. Al contrario que la mayoría de cifrados, MTProto esta enfocado en ser multisesión e independiente de la plataforma y el transporte de archivos independiente de su formato. MTProto tiene dos versiones.

- **MTProto v1.** En esta versión los mensajes son cifrados con el algoritmo *SHA-1*.
- **MTProto v2.** En 2017 se actualizó MTProto, en esta versión se cambió el cifrado *SHA-1* por *SHA-256* debido a una vulnerabilidad encontrada en *SHA-1*. Esta es la versión que se utiliza actualmente.

La información técnica sobre MTProto que se desarrollará a continuación ha sido obtenida de [27] y [9].

7.1.1. Descripción general y resumen de los componentes

MTProto 2.0 es una suite de protocolos criptográficos diseñados para implementar de manera rápida, escalable y segura intercambio de mensajes sin depositar esa responsabilidad en la seguridad del transporte debajo de

dicho protocolo. El protocolo esta subdividido en tres componentes virtuales independientes.

- **Componente de alto nivel:** Define el método por el cual las consultas de la API y las respuestas se convierten en mensajes binarios.
- **Capa criptográfica (autorización):** Define el método por el cual los mensajes están cifrados antes de ser enviados a través del protocolo de transporte.
- **Componente de transporte:** Define el método por el cual el cliente y el servidor transmiten los mensajes sobre otro protocolo de red como HTTP, HTTPS, WS, WSS, TCP o UDP.

Y se pueden resumir como sigue.

Componentes de alto nivel(Lenguajes de consulta/API RPC)

Desde el punto de vista del componente de alto nivel, el cliente y el servidor intercambian mensajes dentro de una sesión.
La sesión se adjunta al cliente en lugar de una conexión *websocket/http/https/tcp.* Además, cada sesión tiene asociada a clave ID de usuario mediante la cual se logra la autorización.
Pueden estar abiertas varias conexiones a un servidor, los mensajes pueden ser enviados en cualquier dirección a través de cualquiera de las conexiones. Cuando se usa el protocolo UDP, una respuesta puede ser devuelta por una dirección de IP distinta.
Hay diferentes tipos de mensajes, estos son.

- **LLamadas RPC(cliente-servidor):** LLamadas a los métodos de la API.
- **Respuestas RPC(servidor-cliente):** Resultados de las llamadas RPC.
- **Notificación del estado de los mensajes.**
- **Consultas de estado de mensaje.**
- **Mensaje multiparte o contenedor.**

Desde el punto de vista de protocolos de bajo nivel, un mensaje es un flujo de datos alineados con 4 ó 16 bytes de límite. Los primeros campos en un mensaje están fijos y son usados por el sistema criptográfico o de autorización.

Cada mensaje, consiste en un identificador del mensaje *(Message Identifier)* de 64 bits, *número de secuencia del mensaje dentro de una sesión*, *longitud* de 32 bits y *cuerpo del mensaje* de cualquier tamaño, siempre y cuando sea múltiplo de 4. Además cuando un contenedor o un mensaje simple se envían, una *cabecera interna* se añade al principio del mensaje, luego el mensaje es cifrado y se le añade una *cabecera externa* la cual será una *clave de identificación* de 64 bits y una *clave del mensaje* de 128 bits.
El *cuerpo* del mensaje normalmente consiste en un *tipo mensaje* de 32 bits seguido de los *parámetros dependientes del tipo*.
Los números están escritos en *little endian*. Sin embargo los números muy grandes (2048 bits) usados en **RSA** y **DH** están escritos en *big endian* porque es lo que hace la biblioteca **OpenSSL**.

Autorización y Cifrado

Antes de que un mensaje sea transmitido por la red usando un protocolo de transporte, este es cifrado añadiendo una cabecera externa la cual es insertada al principio del mensaje y contiene:

- *Key Identifier* de 64 bits,
- *Message Key* de 128 bits.

Una clave de usuario junto con una clave de mensaje definen una clave de 256 bits la cual es la que cifra el mensaje usando un cifrado *AES-256*. La primera parte del mensaje cifrado contiene datos variables (sesión, id del mensaje, número de secuencia) los cuales influyen en la clave del mensaje. La clave del mensaje es definida como los 128 bits iniciales del mensaje cifrado con *SHA-256*, además los mensajes en varias partes, que son los mensajes que se fragmentan debido a su tamaño, están cifrados como un solo mensaje.
Lo primero que tiene que hacer la aplicación cliente es crear una clave de autorización que se genera normalmente la primera vez que se ejecuta la aplicación y por lo general nunca cambia.
Para prevenir potenciales ataques debido a la apropiación de la clave de autorización MTProto soporta *Perfect Forward Secrecy* tanto en los chats en la nube como en los chats secretos.

Sincronización de la hora

Si la hora de un cliente difiere de la hora del servidor, el servidor podría empezar a ignorar los mensajes de este y recíprocamente el cliente a los

mensajes del servidor debido a que el mensaje tenga un indentificador inválido del mensaje.
Bajo estas circunstancias, el servidor enviará un mensaje especial al cliente el cual contendrá la hora correcta, este mensaje será el primero en el caso de que también se envíe un grupo de mensajes.
Habiendo recibido el mensaje, el cliente primero ejecutará una sincronización de la hora y después verificará la clave del mensaje*(Message Key)* para ver si es correcto.
En caso de que no sea correcto, el cliente deberá generar una nueva sesión para asegurar la monotonía de los *Message Keys*.

7.1.2. Descripción de las claves:

En esta sección se describirán las distintas claves que entran en juego en el proceso de cifrado y descifrado de MTProto 2.0 [8].

Authorization Key (auth_key)
Es una clave de 2048 bit compartida por el dispositivo del cliente y el servidor, se crea durante el registro del usuario, se almacena en el dispositivo de este mediante el protocolo de intercambio de claves *Diffie-Hellman* y nunca se transmite a través de la red. Cada *Authorization key* es única y dependiente del usuario, aunque un usuario puede tener más de una ya que Telegram permite tener sesiones persistentes en diferentes dispositivos. En caso de ser necesario estas claves pueden ser bloqueadas para siempre. Por ejemplo podría pasar si un dispositivo con sesión persistente se pierde.

Server Key
Es una clave RSA de 2048 bits usada por el servidor para firmar sus mensajes durante el proceso de registro y la *Authorization Key* todavía no se ha generado. La aplicación tiene una clave pública del servidor que puede ser utilizada para verificar la firmas pero no para firmar mensajes. La clave privada del servidor es almacenada en este y raramente cambia.

Key Identifier (auth_key_id)
Se usan los 64 bits menos significativos del hash *SHA1* de la *Authorization Key* para indicar qué clave en particular se ha usado para cifrar el mensaje. Las claves tienen que ser identificadas unívocamente y en caso de colisión, la *Authorization Key* se regenera. Un identificador Zero Key significa que el cifrado no se usa. Esto está permitido para muy pocos mensajes usados durante el registro para generar la clave en el intercambio *Diffie-Hellman*.

Session

Es un número de 64 bits generado aleatoriamente por el cliente para distinguir entre sesiones individuales como pueden ser diferentes instancias de la aplicación creadas con la misma *Authorization Key*, donde una instancia de la aplicación es la conjunción de la *Key Identifier* y la *Session*.

Bajo ninguna circunstancia un mensaje perteneciente a una sesión puede ser enviado a otra.

Server Salt

Es un número de 64 bits generado aleatoriamente que cambia cada 30 minutos independiente de las sesiones. Se genera por una petición del servidor. Una vez generado el nuevo salt, todos los mensajes tienen que tenerlo. Si bien, se aceptan los mensajes con el salt previo. El *Server Salt* es necesario para proteger ante ciertos ataques como podría ser ajustar el reloj de la víctima en un momento futuro.

Message Identifier (msg_id)

Es un número de 64 bits dependiente del tiempo usado únicamente para identificar mensajes que no tienen asociada una clave de sesión *(Session)*. Los *Message Identifiers* del servidor módulo 4 dan 1 si el mensaje es una respuesta a un mensaje del cliente y dan 3 en otro caso. Los *Message Identifiers* del cliente deben incrementarse monótonamente, igualmente con los del servidor y tienen que ser aproximadamente igual a $unixtime * 2^{32}$, donde *unixtime* es un sistema para la descripción de instantes de tiempo definida como la cantidad de segundos transcurridos desde la medianoche UTC del 1 de enero de 1970. De esta manera, el *Message Identifier* señala el momento aproximado en el que el mensaje fue creado siendo rechazado alrededor de 300 segundos después o 30 segundos antes de ser creado (necesario como medida de protección de ataques de repetición).

Content-related Message

Un mensaje que requiere un reconocimiento explícito. Un reconocimiento explícito es un ack que envía el cliente al servidor para certificar que es él el que envía el mensaje. Esto incluye todos los mensajes de usuario y muchos de servicio, a excepción de contenedores y otros reconocimientos.

Message Sequence Number (msg_seqno)

Un número de 32 bits igual o el doble del número de mensajes *content-related* creados por el remitente antes de este mensaje y posteriormente se va incrementado en uno si el mensaje es del tipo *cotent-related*. Un contenedor se genera siempre después de su contenido por lo que su

Message Sequence Number será siempre igual o mayor a los números de mensajes contenidos en él.

Message Key (msg_key)
En el protocolo **MTProto 2.0**, la *Message Key* se define como los 128 bits del medio del hash *SHA-256* del mensaje que va a ser cifrado antepuesto por un fragmento de 32 bytes de la clave de autorización. En el protocolo **MTProto 1.0**, la *Message Key* se definía como los 128 bits menos significativos del hash *SHA-1* del mensaje a ser cifrado, los bytes de relleno eran excluidos en el cálculo del hash. La *Authorization Key* no estaba involucrada en este cálculo.

Internal (cryptographic) Header
Una cabecera de 16 bytes añadida antes de que el mensaje o el contenedor sea cifrado. Consiste en el *Server Salt* de 64 bits y la *Session* de 64 bits.

External (cryptographic) Header
Una cabecera de 24 bytes que se añade antes de que el mensaje o el contenedor sea cifrado. Consiste en la *auth_key_id* de 64 bits y la *msg_key* de 128 bits.

Payload
Es el *External Header* + mensaje cifrado o contenedor.

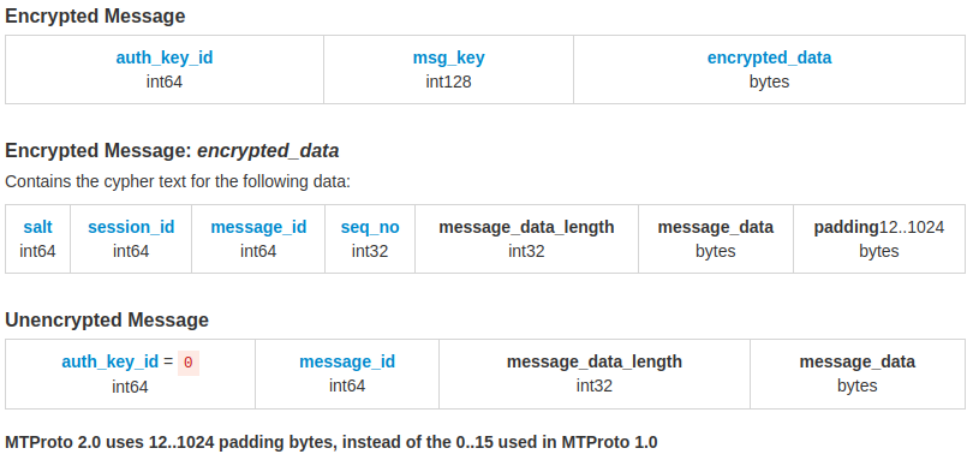


Figura 7.1: Mensajes de MTProto [9].

7.1.3. Creación de la *Authorization Key*

Como hemos visto en el apartado anterior la *Authorization Key* se genera durante el registro del usuario en la aplicación. El formato de las consultas usa *Binary Data Serialization*, que es un proceso en el cual se convierten los datos en binario, ya que MTProto requiere que los tipos de datos estén en formato binario y *TL Language* que es un lenguaje de tipos usado para describir constructores, tipos y funciones existentes.
Los números de gran tamaño son transmitidos como cadenas que contienen la secuencias de bytes en formato *big endian*, los números de menor tamaño como pueden ser los *int, long int128...* usan normalmente el formato *little endian*. Si pertenecen al hash generado con *SHA-1* los bytes no son reorganizados.
Una vez introducidos los formatos que seguirán las consultas y los números veamos los pasos que se siguen en la creación de la *Authorization Key*.

1. El cliente envía una consulta al servidor, en esta consulta irá el *nonce* que es un número aleatorio *int128* generado por el cliente que servirá para para que el servidor lo identifique, este número no es secreto y a partir de ese momento irá incorporado en todas las consultas. Con este paso empieza el intercambio de claves *Diffie-Hellman*.

2. El servidor le responde enviando

   - *server_nonce*: Es un número aleatorio *int128* generado por el servidor que sirve para que el cliente lo identifique y al igual que el *nonce* no será secreto e irá incluido en las siguientes consultas y respuestas.
   - *pq*: Es una representación de un número natural en formato *big endian* que es el producto de dos números primos, *pq* por lo general verifica $pq \leq 2^{63} - 1$
   - *server_public_key_fingerprints*: Es una lista de *fingerprints*, secuencias únicas de letras y números usadas para identificar las claves RSA públicas.

3. El cliente descompone *pq* en factores primos tal que $p<q$. Una vez hecho esto empieza el intercambio de claves Diffie-Hellman. Este paso es la prueba de trabajo.

4. En este paso se presentan las pruebas de trabajo. Para ello se genera *encrypted_data* como sigue.

   a) Se genera *new_nonce* un número aleatorio generado por el cliente.

   b) Se genera *data* que es una serialización de:

   - *pq*,

- p,
- q,
- *server_nonce*,
- *new_nonce*,
- *dc*, que es la id del servidor con el que se está haciendo el intercambio de claves.

c) Se genera *encrypted_data* como el resultado de la función $RSA\_PAD(data, server\_public\_key)$. Esta es una variación de RSA cuyo funcionamiento es el siguiente.

- Se genera *data_with_padding:=data+random_padding_bytes*, donde *random_padding_bytes* se escoge de manera que *data_with_padding* tenga un tamaño final de 192 bytes.
- Luego se crea la variable *data_pad_reversed* como resultado de la función $BYTE\_REVERSE(data\_with\_padding)$, función la cual revierte el orden de los bytes.
- Se genera un número aleatorio de 32 bytes que se denomina como *temp_key*.
- Se genera $data\_with\_hash := data\_pad\_reversed + {} + SHA256(temp\_key + data\_with\_padding)$. Después de esta operación *data_with_hash* tiene exactamente una longitud de 224 bytes.
- Después de este paso se encripta *data_with_hash* a partir de *temp_key* usando AES-256 y se almacena el resultado en la variable *aes_encrypted*.
- Se ajusta *aes_encrypted* a 32 bytes usando la función SHA-256 y a continuación se realiza la operación XOR con *temp_key* almacenándose el resultado en *temp_key_xor*.
- Se combinan *temp_key_xor* con *aes_encrypted* y se almacena el resultado en *key_aes_encrypted*. Esta tiene un tamaño de 256 bytes.
- Se compara el tamaño *key_aes_encrypted* con *RSA-modulus* de *server_pub_key* en formato big endian de 2048 bits. Si *key_aes_encrypted* es mayor que *RSA-modulus*, se empieza de nuevo el proceso eligiendo un nuevo *random_temp_key*.
- Por último se genera *encrypted_data* como resultado de aplicar RSA a *key_aes_encrypted* y a *server_pub_key*.

Una vez hecho esto, se envía *encrypted_data* al servidor. Después de este paso alguien podría interceptar la consulta y modificarla con una cosulta suya haciendo un ataque **man-in-the-middle**. Este ataque no sería muy efectivo, ya que el único elemento que podría modificar sería *new_nonce* porque los demás están cifrados y el resultado sería que el

atacante genere una *Authorization_key* propia independiente de la del cliente haciendo que el ataque no sea efectivo.

5. El servidor responde enviando

   - *nonce*.
   - *server_nonce*.
   - *encrypted_answer*: Respuesta cifrada que es del tipo *string* y a su vez contiene
     - *new_nonce_hash*: Son los 128 bits menos significativos de *SHA-1(new_nonce)*.
     - *answer*: Es una serialización de *nonce*, *server_nonce*, *g*, *dh_prime*, *g_a* y *server_time*. Donde *dh_prime* es un número primo seguro de 2048 bits, *g* un generador de un subgrupo cíclico de orden $\frac{p-1}{2}$, g_a es la clave que envía el servidor al cliente que al elevarla a *b* módulo *dh_prime* el cliente obtendría la clave final.
     - *answer_with_hash*: Es una generación con la función hash *HASH1* quedando como *SHA-1(answer)+answer+(0-15 bytes aleatorios)* de manera que la longitud sea divisible por 16, es del tipo *string*.
     - *answer_aes_key*: *SHA-1(new_nonce + server_nonce) + + substr(SHA-1(server_nonce + new_nonce), 0, 12)* y es del tipo *string*. Donde *substr* es una función que devuelve caracteres del *string* desde una posición hasta otra, en este caso desde la posición 0 hasta 12.
     - *tmp_aes_iv*: *substr(SHA-1(server_nonce + new_nonce), 12, 8) + SHA-1(new_nonce + new_nonce) + substr(new_, 0, 4)*
     - *encrypted_answer*: *AES256_ige_encrypt(answer_with_hash, tmp_aes_key, tmp_aes_iv)* donde:
       - *tmp_aes_key*: Es una clave de 256 bits
       - *tmp_aes_iv*: Es un vector de inicialización de 256 bits.

       Al igual que en el resto de las instancias que usan el cifrado AES, a los datos cifrados se le añaden bytes aleatorios de forma que el tamaño sea divisible por 16.

Después de este paso *new_nonce* sigue siendo únicamente conocido por el cliente y el servidor de esta manera el cliente garantiza que el servidor es el que está al otro lado de la comunicación y que la respuesta de este es correcta, ya que los datos están cifrados usando *new_nonce*.
El cliente comprueba que $p = dh\_prime$, es un número primo seguro de 2048 bits, es decir, se tiene que verificar que p y $\frac{p-1}{2}$ son primos, además, $2^{2047} < p < 2^{2048}$, y g genera un subgrupo cíclico con orden

primo $\frac{p-1}{2}$.g siempre vale 2, 3, 4, 5, 6, o 7.
Si la verificación tarda mucho tiempo, cosa que ocurre en dispositivos antiguos, se ejecutarían solo 15 iteraciones en el algoritmo de Miller-Rabin para garantizar que p y $\frac{p-1}{2}$ sean primos con una probabilidad de error muy baja, alrededor de una millonésima, y dejar el resto de iteraciones para después, ejecutándose estas de fondo.

6. El cliente genera un número aleatorio b de 2048 bits y lo envía al servidor en un mensaje que contiene:
   - *nonce*.
   - *server_nonce*.
   - *encrypted_data* que se descifra de la siguiente manera:
     - $g\_b = g^b$ mod *dh_prime*
     - *data* que es una serialización donde
       - *nonce*
       - *server_nonce*
       - *retry_id* que vale 0 en el primer intento y en caso contrario, vale *auth_key_aux_hash* del intento fallido anterior y es del tipo *long*.
       - *g_b* que es del tipo *string*.
     - *data_with_hash* que es: *SHA-1(data)+data+(0-15 bytes aleatorios de manera que el tamaño sea divisible por 16)*.
     - *encrypted_data* que es: *AES256_ige_encrypt(data_with_hash, tmp_aes_key, tmp_aes_iv)*. Donde el modo IGE es una variación de modo CBC [2.2] de los cifrados de bloque. En este modo se garantiza que si un bloque del mensaje encriptado es cambiado, no se pueda desencriptar correctamente el mensaje completo.

7. Una vez hecho los pasos previos tendríamos que *auth_key* vale g^{ab} mod *dh_prime*, en el servidor se calcula como g_b^a *mod* dh_prime y en el cliente se calcula como g_a^b mod *dh_prime*.

8. *auth_key_hash* se calcula como los 64 bits de menor prioridad de *SHA-1(auth_key)*. El servidor comprueba si existe alguna otra clave con el mismo *auth_hash* y responde de alguna de las siguientes tres formas

   a) Una serialización de:
   - *nonce*,
   - *server_nonce*,
   - *new_nonce_hash1*.

   b) Una serialización de:
   - *nonce*,

- *server_nonce*,
- *new_nonce_hash2*.

c) Una serialización de:

- *nonce*,
- *server_nonce*,
- *new_nonce_hash3*.

Donde *new_nonce_hash1*, *new_nonce_hash2* y *new_nonce_hash3* son los 128 bits menos significativos de SHA-1 de la cadena de bytes obtenida al añadir a *new_nonce* un byte con el valor 1,2 o 3 respectivamente y seguido de *auth_key_hash*.
Auth_key_aux_hash son los 64 bits más significativos del resultado de la función hash *SHA-1(auth_key)*.
Si algo falla durante estos pasos, el cliente volvería al paso 6 y generándose un nuevo *b*. Al mismo tiempo se define *server_salt* como *substr(new_nonce, 0, 8)* XOR *substr(server_nonce, 0, 8)*.

Gestión de errores

Si el cliente no obtiene alguna respuesta del servidor en un intervalo de tiempo determinado se repite la consulta, análogamente ocurre con el servidor. Sin embargo si el servidor no obtiene una segunda respuesta del cliente en 10 minutos, reiniciará la conexión y el cliente tendrá que empezar de nuevo.

7.1.4. Generando la clave y el vector de inicialización de AES

En esta sección hablaré de como se generan la clave de autorización (*auth_key*) y de la clave del mensaje (*msg_key*) necesarias para calcular la clave de AES (*aes_key*) y el vector de inicialización de 256 bits (*iv_aes*) usados para cifrar los mensajes en MTProto 2.0.
El algoritmo sigue los siguientes pasos.

1. Calculamos *msg_key_large* como *SHA-256(substr(auth_key, 88+x, 32)+plaintext+random_padding)*.

2. Calculamos *msg_key* como *substr(msg_key_large, 8, 16)*.

3. Calculamos *sha256_a* como *SHA-256(msg_key+substr(auth_key, x, 36))*.

4. Calculamos *sha256_b* como *SHA-256(substr(auth_key, 40+x, 36)+msg_key)*.

Y una vez hechos estos pasos, ya podemos calcular la clave para AES y el vector de inicialización.

- ***aes_key***: *substr(sha256_a, 0, 8)+substr(sha256_b, 8, 16)+ +substr(sha256_a, 24, 8).*

- ***aes_iv***: *substr(sha256_b, 0, 8) + substr(sha256_a, 8 16)+ +substr(sha256_b, 24, 8).*

x vale 0 cuando los mensajes van del cliente al servidor y 8 cuando los mensajes van del servidor al cliente.
Los 1024 bits menos significativos de la *auth_key* no se utilizan para el cálculo ya que estos se usan para cifrar la copia local de los datos recibidos del servidor además, los 512 bits menos significativos no se almacenan en el servidor por lo que si el cliente pierde la clave o la contraseña del dispositivo, no se podrán descifrar los datos locales. Un esquema del proceso se puede ver en 7.2.

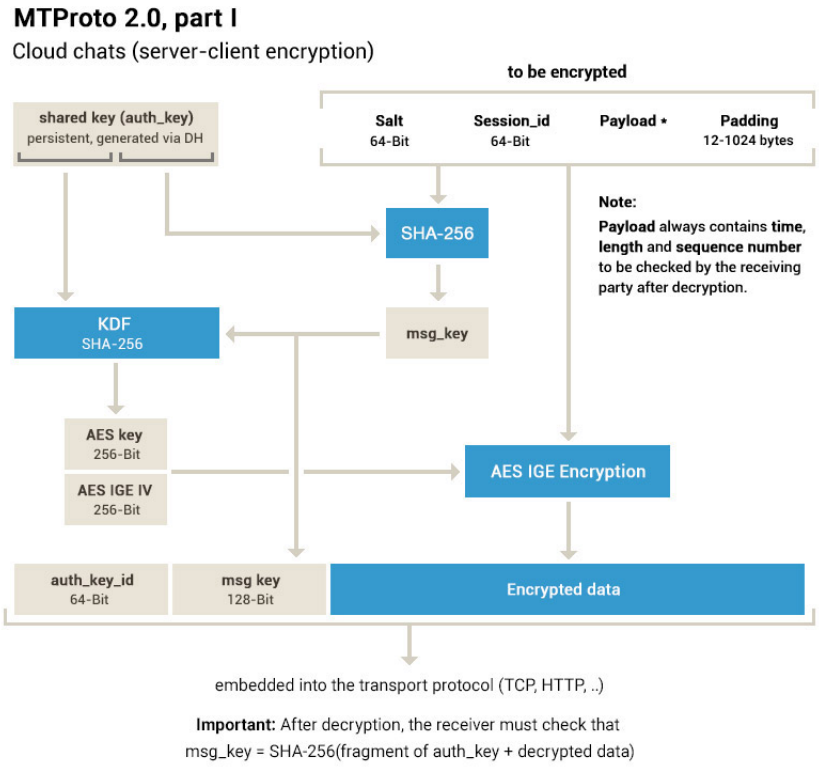


Figura 7.2: Esquema del cifrado de mensajes usado en MTProto 2.0 [9].

7.1.5. Envío de mensajes

Una vez realizado el intercambio de claves mediante *Diffie-Hellman* y la generación de la clave y el vector de inicialización de AES ya se podrían enviar mensajes cifrados entre el cliente y el servidor utilizando *AES256*. Los protocolos de transporte que están disponibles son los siguientes.

- *TCP*
- *WebSocket*
- *WebSocket* sobre *HTTPS*
- *HTTP*
- *HTTPS*
- *UDP*

7.2. WhatsApp, Facebook Menssenger y Signal (TextSecure Protocol)

El protocolo *TextSecure Protocol* también conocido como *Signal* fue desarrollado por Trevor Perrin y Moxie Marilinspike que trabajaban en la empresa Open Whisper Systems en 2013. Se implementó inicialmente en la aplicación de mensajería homónima *Signal* aunque posteriormente se introdujo en otras aplicaciones de mensajería como *WhatsApp* y *Facebook Menssenger*.
Este protocolo ha tenido 3 versiones.

- **TextSecure v1.** Es la primera versión del protocolo, que fue lanzada en 2013 y estaba basada en el protocolo *Off-the-Record Messaging (OTR)*.
- **TextSecure v2.** Es la segunda versión publicada el 24 de febrero de 2014. Esta versión extendió el protocolo con *Double Ratchet Algorithm* el cual un intercambio de claves *Diffie-Hellman* con una *función de derivación de clave (KDF)* usando funciones hash, permitiendo extender las claves y aumentando la seguridad del protocolo.
- **TextSecure v3.** Esta tercera versión fue lanzada en octubre de 2014 y añadió algunos cambios a las primitivas criptográficas y al protocolo de red.

Cabe a mencionar que aunque el protocolo originalmente el protocolo se llamaba *TextSecure* en 2016 fue renombrado como *Signal Protocol* que es como lo conocemos hoy en día.
La información técnica de este protocolo que voy a desarrollar a continuación ha sido obtenida de [15].

7.2.1. Descripción general y dispositivos

El protocolo *Signal* es un protocolo diseñado para prevenir que aplicaciones como *WhatsApp, Signal y Facebook Messenger* sean vulneradas de manera que se pueda acceder a la información intercambiada en los mensajes y las llamadas. Este protocolo permite que un usuario tener diversos dispositivos cada uno con su propias claves garantizando que si se obtiene alguna de ellas, los mensajes enviados por alguno de los otros dispositivos no puedan ser descifrados. Además también es usado en el caso de *WhatsApp* para cifrar el historial de mensajes y enviarlo a un nuevo dispositivo del mismo usuario.
Por comodidad voy a explicar el protocolo para WhatsApp dado que para el resto de aplicaciones que lo usan hacen un uso igual de este.

Tipo de dispositivos

Como he me mencionado anteriormente, el protocolo *Signal* permite tener varios dispositivos asociados al usuario, si bien no todos los dispositivo son iguales. Se pueden distinguir dos tipos de dispositivos.

- **Dispositivo principal.** Dispositivo único utilizado para vincular una cuenta de *WhatsApp* con un número de teléfono. Este dispositivo permite vincular dispositivos adicionales que serán los dispositivos compañeros.
- **Dispositivo Compañero o Secundario.** Es un dispositivo vinculado a una cuenta existente de WhatsApp, a diferencia del dispositivo principal, este no tiene porque ser único.

Ciertas aplicaciones únicamente pueden ser usadas en dispositivos principales como pueden ser las aplicaciones para *Android* y *iOS*.

7.2.2. Descripción de las claves

En este apartado hablaré acerca de las distintas claves que se utilizan para cifrar y descifrar los datos.

- **Claves públicas**
  - *Identity Key Pair*: Es un par de claves de largo plazo del tipo *Curve25519* generadas en la instalación.
  - *Signed Pre Key*: Es una clave de medio plazo del tipo *Curve25519* generada durante la instalación y firmada por la Identity Key que se va rotando de manera periódica a lo largo del tiempo.
  - *One-Time Pre Keys*: Es un lote de pares de claves del tipo *Curve25519* de un solo uso generadas en la instalación y siendo posible volver a generarlas en caso de ser necesario.
- **Claves de Sesión**
  - *Root Key*: Clave de 32 bytes usada para generar *Chain Key*.
  - *Chain Key*: Clave de 32 bytes usada para generar *Message Key*.
  - *Message Key*: Clave de 80 bytes usada para cifrar los mensajes. Están formadas por 32 bytes usados para la clave *AES256*, 32 bytes para la clave *HMAC-SHA256* y 16 bytes para un *IV*.
- **Otras claves**
  - *Linking Secret Key*: Es una clave de 32 bytes generada en un *dispositivo compañero* y que tiene que ser enviada por un canal seguro al *dispositivo principal*. Se usa para verificar un *HMAC* del intercambio durante la vinculación entre los dispositivos. Se envía escaneando un código QR.

7.2.3. Otros elementos relacionados con los dispositivos compañeros

- *Linking Metadata*: cifrado de metadatos asignados a un dispositivo compañero durante la etapa de vinculación, se usa a la par que la *Identity Key* para identificar un dispositivo compañero entre los dispositivos de WhatsApp.
- *Signed Device List Data*: Lista cifrada que identifica los dispositivos compañeros vinculados a la cuenta principal en el momento de la firma. Se firma con la *Identity Key* del dispositivo principal usando 0x0602 como prefijo.
- *Account Signature*: Firma del tipo *Curve25519* calculada a partir del prefijo 0x600, *Linking Metadata* y la *Identity Key* del dispositivo compañero usando la *Identity Key* del dispositivo principal.

- *Device Signature*: Firma del tipo *Curve25519* calculada a partir del prefijo 0x601, *Linking Metadata*, la *Identity Key* del dispositivo compañero y la *Identity Key* del dispositivo principal usando la *Identity Key* del dispositivo compañero.

Una vez introducido la terminología que se va a usar el la descripción del protocolo *Signal*, procederé a describir las etapas principales de este.

7.2.4. Registro de clientes

Etapa inicial consistente en añadir dispositivos asociados al cliente, como se ha visto anteriormente existen dos tipos de dispositivos: *dispositivo principal* y *dispositivos compañeros* y en función del tipo se registrarán de manera diferente.

Dispositivo principal

Durante el momento del registro, el cliente de WhatsApp envía al servidor su *Identity Key*, su *Signed Pre Key* con una firma y conjunto de *One-Time Pre Keys*. Una vez enviadas el servidor de WhatsApp almacena estas estas claves públicas asociadas con el indentificador del usuario.

Dispositivo compañero

Para vincular un nuevo dispositivo a la cuenta de WhatsApp, el dispositivo principal del usuario crea al principio una firma de la cuenta firmando la *Identity Key* del nuevo dispositivo, a su vez, el dispositivo compañero que se quiere introducir firmando la *Identity Key* pública del dispositivo principal. Una vez que se han realizado ambas firmas, ya se puede iniciar las sesión con el dispositivo compañero usando un cifrado *end-to-end*.
Los pasos seguidos son:

1. El dispositivo muestra su *Identity Key* ($I_{companion}$) y genera un una clave temporal para vincularse ($L_{companion}$) en un código QR. Esta clave nunca será enviada al servidor.
2. El dispositivo principal escanea el código QR y almacena $I_{companion}$ en el disco.
3. El dispositivo principal carga su propia *Identity Key* como $I_{primary}$.
4. El dispositivo genera los metadatos de la vinculación ($L_{metadata}$) y actualiza la lista de datos de dispositivos para que contenga un nuevo dispositivo compañero como ($L_{istData}$).

5. El dispositivo principal genera una firma de la cuenta para el compañero,

   $A_{signature} = CURVE25519\_SIGN(I_{primary}||L_{metadata}||I_{companion})$.

6. El dispositivo principal genera una firma para la lista de dispositivos que permitirá actualizar la propia lista de dispositivos.

   $ListSignature = CURVE25519\_SIGN(I_{primary}, 0x0602||ListData)$.

7. El dispositivo principal agrupa los datos de la vinculación en L_{data}, conteniendo este $L_{metadata}$, $I_{primary}$ y $A_{signature}$.
8. El dispositivo principal genera una $HMAC$ para la vinculación y una $PHMAC$ que será igual a $HMAC - SHA256(L_{companion}, L_{data})$. Una vez hecho esto enviará *ListData*, *ListSignature*, L_{data} y la $PHMAC$ al servidor.
9. El servidor almacena *ListData* y *ListSignature* y reenvía L_{data} y la $PHMAC$ al dispositivo compañero.
10. El dispositivo compañero verfica la $PHMAC$, decodifica *Ldata* en *Lmetadata*, *Iprimary* y *Asignature* verificando esta última.
11. El dispositivo compañero almacena $L_{metadata}$ e $I_{primary}$ en el disco.
12. A continuación este genera una firma del dispositivo por si mismo que es de la forma
    $D_{signature} = CURVE25519\_SIGN(I_{companion}, 0x0601||$
    $||L_{metadata}||I_{companion}||I_{primary})$.
13. El dispositivo compañero sube al servidor de WhatsApp: $L_{metadata}$, $A_{signature}$, $D_{signature}$, $I_{companion}$, la *Signed Pre Key* pública del dispositivo firmada y un lote de *Pre Keys* de un solo uso.
14. El servidor almacena los datos subidos asociados con la identificación del usuario combinados con el identificador especifico del dispositivo.

7.2.5. Inicio de sesión

Para que la comunicación entre usuarios sea segura y privada el emisor establece una conexión por pares con cada uno de los dispositivos del receptor. Una vez que la conexión entre emisor y receptor ha sido establecida, no es necesario volver a establecerla a no ser que la sesión se pierda.
Lo pasos que se siguen para establecer una conexión son los siguientes.

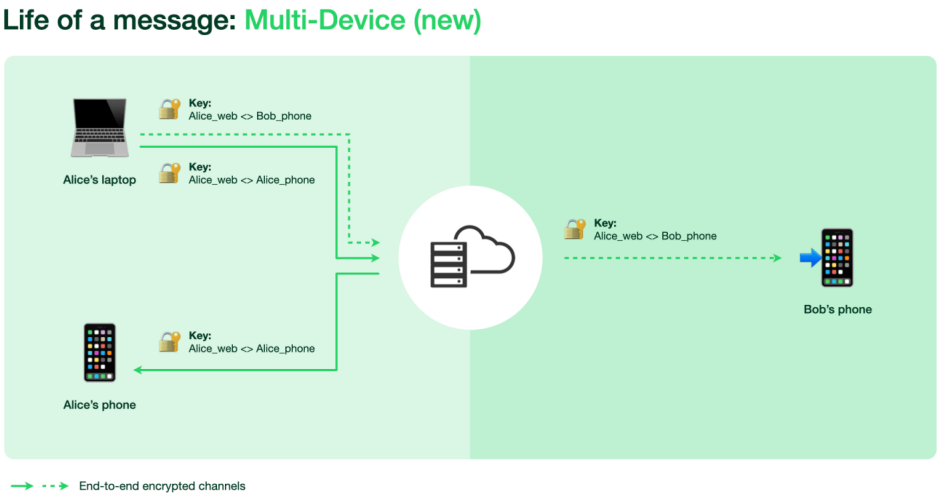


Figura 7.3: Diagrama de un conexión entre dos dispositivos teniendo uno de ellos un dispositivo compañero [7].

1. El cliente que inicia la conexión solicita la *identity key*, la *Signed Pre key* y un lote de *Pre keys* de un solo uso para cada dispositivo del receptor y los dispositivos adicionales que el mismo posee excluyendo el dispositivo desde el que se inicia la conexión.

2. El servidor devuelve todo lo solicitado y elimina las *Pre keys* enviadas ya que son de un solo uso. Sí el último lote de *Pre keys* enviado es agotado y el receptor no los ha repuesto no se devuelve ninguna clave. Además, por cada dispositivo compañero que haya tanto del emisor como del receptor, el servidor devuelve $L_{metadata}$, $A_{signature}$ y $D_{signature}$ que fueron enviadas por el dispositivo compañero cuando se vinculó.

3. Por cada conjunto de claves devueltas, el emisor tiene que verificar $Asignature$ con $CURVE25519\_VERIFY\_SIGNATURE(I_{primary}, 0x0600||L_{metadata}||I_{companion})$.
$D_{signature}$ es obtenida con $CURVE25519\_VERIFY\_SIGNATURE(I_{companion}, 0x0601||L_{metadata}||I_{companion}||I_{primary})$. Si en algún momento la verificación falla, el emisor termina la sesión de cifrado y no envía ningún mensaje al dispositivo que ha fallado.

Una vez obtenidas las claves del servidor y se ha verificado la identidad, el emisor inicia la sesión de encriptación con cada dispositivo individualmente. Para ello se siguen los siguientes pasos:

1. El emisor almacena la *Identity Key* del receptor como $I_{recipient}$, la *Signed Pre Key* como $S_{recipient}$ y la *Pre Key* de un solo uso como $O_{recipient}$.
2. El emisor genera un par de pares de claves efímeras *Curve25519* llamada $E_{initiator}$.
3. El emisor carga su propia *Identity Key* como $I_{initiator}$.
4. El emisor calcula el *master_secret* como
$master\_secret = ECDH(I_{initiator}, S_{recipient})||ECDH(E_{initiator}, I_{recipient})||$
$||ECDH(E_{initiator}, S_{recipient})||ECDH(E_{initiator}, O_{recipient})$.
Cabe a mencionar que *ECDH* es el intercambio de claves *Diffie-Hellman*.
5. El emisor usa *HKDF* para crear una *Root Key* y una *Chain Key* de *master_secret*. Donde HKDF es una función de derivación de claves simple basada en el código de autenticación de mensajes *HMAC*[21].

7.2.6. Intercambio de mensajes

Una vez que la sesión se ha establecido, los clientes intercambian los mensajes encriptados con *Message Key* usando *AES-256* con el modo *CBC*[2.2] y para la autenticación *HMAC-SHA256*. *Message Key* cambia con cada mensaje que se envía y además es efímera para que esta no pueda ser reconstruida una vez que el mensaje sea transmitido y recibido. Esta clave se obtiene a partir de la *Chain Key* del receptor que se regenera con cada intercambio de mensajes.

7.2.7. Cálculo de *Message Key* a partir de *Chain Key*

La *Message Key* se genera:

1. $MessageKey = HMAC\text{-}SHA256(ChainKey, 0x01)$.
2. La *Chain Key* es actualizada como
$ChainKey = HMAC\text{-}SHA256(ChainKey, 0x02)$.

Este último paso hace que la *Chain key* cambie haciendo imposible que con una Message Key antigua se obtenga la *Chain Key* actual.

7.2.8. Cálculo de *Chain Key* a partir de *Root Key*

Cada vez que un mensaje es enviada una clave pública efímera *Curve25519*. Una vez que la respuesta es definida se calcula una nueva *Chain Key* y una nueva *Root Key* de la siguiente forma:

1. $ephemeral\_secret = ECDH(Ephemeral_{sender}, Ephemeral_{recipient})$.
2. $Chain\ Key, Root\ Key = HKDF(Root\ Key, ephemeral\_secret)$.

7.3. iMessage

iMessage es una aplicación de mensajería diseñada para ser utilizada en dispositivos iOS, iPadOS, Apple Watch y ordenadores con MacOS. Esta aplicación permite enviar mensajes de texto, y archivos como pueden ser fotos, contactos, ubicaciones etc. La información para este apartado ha sido obtenido mayormente de [4] y [3].

Funcionamiento criptográfico

Para iniciar una nueva conversación a partir de una dirección o un nombre. En caso de usar un número de teléfono o una dirección de correo electrónico, la aplicación contacta con el servicio de identidad de Apple (*IDS*). Este es un directorio de claves públicas de iMessage, direcciones del servicio de notificaciones push de Apple (APNs), números de teléfono y direcciones de correo electrónico usadas para obtener las claves y direcciones de los dispositivos [11]. Una vez que se pone en contacto con este se obtienen las claves públicas y las direcciones APNs de todos los dispositivos asociados al destinatario.
Una vez obtenidas las claves, el mensaje que se envía es encriptado de manera individual para cada uno de los dispositivos del destinatario. Tanto las claves públicas usadas como las clases de firmas se obtienen del IDS. El emisor genera un valor aleatorio de 88bits para cada dispositivo de destino del destinatario y este es utilizado como clave *HMAC-SHA256* para crear una valor de 40 bits a partir de la clave pública del emisor, receptor y del texto sin formato. Los 88 bits generados aleatoriamente se combinan con los 40 bits obtenidos para formar una clave de 128 bits para encriptar el mensaje usando el estándar AES en modo contador (CTR).
El dispositivo receptor usa el valor de 40 bits para comprobar la integridad del texto sin formato una vez que este es desencriptado.
Para encriptar esta clave AES se usa REA-OAEP para la clave pública con el dispositivo receptor. Una vez encriptados el mensaje y la clave se genera un hash SHA1 el cual es firmado con el algoritmo de firma digital de curva elíptica ECDSA, para esto también se usa la clave de firma privada del emisor. En dispositivos más recientes que tengan un sistema operativo iOS13 o posterior y iPadOS 13.1 o posterior se puede utilizar ECIES en lugar de RSA.
Los mensajes obtenidos después del proceso encriptación contienen:

- Texto del mensaje encriptado.
- Clave del mensaje encriptada.
- Firma digital del emisor.

Para enviar los mensajes se mandan al APNs, donde los metadatos asociados a este como son la fecha o la información sobre el enrutamiento del APNs no son encriptados. Para comunicarse con el APNs se una comunicación encriptada a través de un canal TLS de secreto-hacia-delante.
Como el APNs solo puede emitir mensajes de 4 o 16KB como máximo dependiendo de la versión del sistema operativo. Por lo que si el mensaje tiene un tamaño más grande como podría pasar si se envía una foto o un vídeo, lo que se hace es encriptar el archivo adjunto con AES en modo CTR utilizando una clave de 256 bits generada aleatoriamente y se carga en iCloud. Una vez subido el archivo encriptado se envía al receptor la clave AES del archivo adjunto, su identificador uniforme de recursos(URI) y un hash SHA1 de su forma encriptada.
En el siguiente esquema se puede ver resumido el proceso de encriptado y desencriptado de los mensajes.

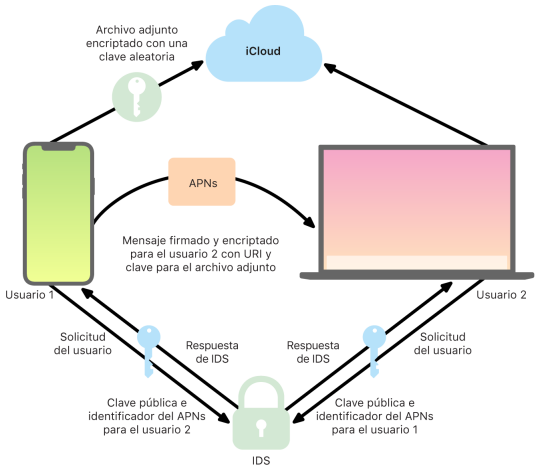


Figura 7.4: Esquema de cifrado en iMessage [3].

Al igual que en las notificaciones push, los mensajes son eliminados del APNs una vez que se envían. Si es verdad que a diferencia de otras notificaciones que se envían del APNs, los mensajes se guardan en una cola para enviarlos a dispositivos sin conexión. Esto permite almacenar los mensajes en un plazo de 30 días máximo permitiendo así que un mensaje llegue tiempo después a un dispositivo que esta sin conexión.

7.4. Line Messenger (Letter Sealing)

Line es una aplicación de mensajería diseñada para ser utilizadas en dispositivos móviles y ordenadores independientemente del sistema operativo. Fue diseñada en 2011 por la empresa surcoreana NHN en Japón. Apareció con la finalidad de paliar la caída de los servicios de telefonía en Japón debido al terremoto. Utiliza un protocolo criptográfico criptográfico propio llamado *Letter Sealing*.
Este protocolo apareció en 2016 y en la actualidad cuenta con dos versiones.

- **Letter Sealing v1**. Fue desarrollado en 2016 y garantiza una protección extremo a extremo en mensajes individuales y en grupo.

- **Letter Sealing v2**. Fue desarrollado en 2019. Gracias a un *Bug Bounty Program*, iniciativa por la cual se recompensa a las personas que descubren y notifican errores de software, desarrollado por Takanori Isobe y Kazuhiko Minematsu se encontraron una serie de vulnerabilidades que aunque en ese momento no eran explotables, en un futuro si podrían llegar a serlo. Por ello se decidió desarrollar una nueva versión de este que garantizara una mayor seguridad.

La información para esta sección ha sido obtenida de [18].

	v1	**v2**
Intercambio de claves	ECDH sobre Curve255195	
Algoritmo para encriptar mensajes	AES256-CBC	AES256-GCM6
Función Hash del mensaje	SHA-256	N/A
Autenticación de los mensajes	AES-ECB con SHA-256 MAC	AES256-GCM

Cuadro 7.1: Tabla comparativa de versiones de Letter Sealing.

7.4.1. Generación de claves

Al igual que en las aplicaciones de mensajería vistas, en *Letter Sealing* se generan un par de claves por cada dispositivo del usuario. Si no existe ninguna en el dispositivo, se genera un par y se almacenan de manera segura en el almacenamiento privado de la aplicación.
Una vez generadas la clave pública se registra en un servidor de LINE, el servidor la asocia la clave al usuario de LINE y le devuelve un *ID* de clave único. Este *ID* representa la última versión de clave pública del usuario.

7.4.2. Intercambio de claves

Para poder intercambiar mensajes utilizando *AES-256* los usuarios tienen que intercambiar previamente una clave secreta para poder cifrar los mensajes. Para ello se utiliza el intercambio de claves *Diffie-Hellman* sobre la familia de curvas *curve25519* como hemos visto en el capítulo 4.
Todo el proceso es transparente para los usuarios y no se necesita de otro elemento para verificar las claves públicas. Esto se debe a que LINE permite a los usuarios ver las huellas de sus recipientes de clave pública.

7.4.3. Cifrado de mensajes

El funcionamiento de cifrado de mensajes en *Letter Sealing v2* es el siguiente.

1. Se deriva una clave para el cifrado usando dos valores, la clave secreta intercambiada durante el intercambio de claves (*SharedSecret*) y valor aleatorio de 16 bytes generado aleatoriamente.

$$salt = random_{secure}(16).$$

$$Key_{encrypt} = SHA256(SharedSecret||salt||"Key").$$

2. Se calcula *nonce* concadenando 8 bytes por cada contador de chat ($per\_chat_{counter}$) con 4 bytes generados aleatoriamente.

$$nonce[12] = per\_chat_{counter}[8]||random_{secure}(4).$$

3. Se encripta el payload (M) del mensaje usando *AES256-GCM*, usando $Key_{encrypt}$ y *nonce*. En *Letter Sealing v2* se añade unos metadatos al mensaje (AAD) para reforzar la integridad del mensaje. Se calcula de la siguiente manera.

$$ADD =$$

$$recipientID||senderID||senderkeyID||recipientkeyID||version||contenttype.$$

   También se utiliza una función llamada *GCM* que es un esquema *AEAD*, *Authenticated Encryption with Associated Data* que garantiza la integridad y confidencialidad del mensaje. La salida del cifrado sería la siguiente.

$$(C, tag) = AES - GCM(Key_{encrypt}, nonce, M, AAD).$$

4. Una vez seguidos estos pasos el emisor envía al receptor un mensaje que consta de los siguientes elementos.

- *version*,
- *content type*,
- *salt*,
- $C||tag$,
- *nonce*,
- *sender key ID*,
- *recipient key ID*.

Donde

Version, Content type sirven para indentificar la version de *Letter Sealing* que se ha utilizado.

Sender Key ID es utilizada por el receptor del mensaje para recuperar la clave pública usada en el cifrado del mensaje.

Recipient key ID es utilizada para verificar que el mensaje puede se desencriptado usando la clave privada actual.

7.4.4. Descifrado de mensajes

Una vez recibido el mensaje, el receptor lo desencripta siguiendo los siguientes pasos.

1. Deriva una clave de cifrado usando *SharedSecret* compartido previamente y *Salt* del mensaje.
2. Descifra el mensaje usando *AES-GCM*.
3. Aporta metadatos al mensaje como *AAD* (*Additional Uthenticated Data*).
4. Si la etiqueta, *tag*, coincide con la que se ha generado desencriptando el mensaje, el receptor imprime el mensaje. Si no coincide, el receptor como medida de seguridad no lo imprime.

Capítulo 8

Implementación de una aplicación de mensajería

En este capítulo voy a explicar como he desarrollado una aplicación de mensajería utilizando las herramientas vistas en la memoria.
Para desarrollar la aplicación he usado el lenguaje de programación **Python**. Aunque este lenguaje no es muy eficiente y consume muchos recursos, para la aplicación que he desarrollado no es necesario esto, ya que son dos programas independientes y ninguno de los dos necesita muchos recursos. Al igual que las aplicaciones vistas en el capítulo anterior, se ha seguido una arquitectura **cliente-servidor** por lo que se han diseñado dos aplicaciones distintas que se comunican entre si usando **sockets**. Una es la aplicación **Servidor**, esta no consta de interfaz gráfica y solo se tiene que ejecutar en un dispositivo. La otra aplicación es la aplicación **Cliente**, esta si consta de interfaz gráfica ya que es la que se va a utilizar como medio para escribir y leer los mensajes. Esta aplicación la tienen que ejecutar todos los usuarios que quieran usar el chat.

8.1. Herramientas utilizadas

Las herramientas que he utilizado para desarrollar la aplicación han sido las siguientes.

- **Gestor de dependencias:** El gestor de tareas que he utilizado ha sido **Poetry**. Este gestor es de los más usado en Python. Está muy bien documentado y como archivo de configuración utiliza un archivo del tipo *.toml* por lo que es muy fácil de configurar.
- **Gestor de tareas:** El gestor de tareas que he usado ha sido **Poe the Poet**. Es un gestor de tareas de reciente aparición y permite

automatizar las distintas tareas de una manera sencilla y con una fácil integración con Poetry. Me he decantado por este ya que esta muy actualizado, tiene una documentación exhaustiva y consta con una comunidad muy amplia.

- **Tests runner:** Para ejecutar tests he usado **Pytest**. Pytest es un framework que utiliza una sintaxis muy sencilla para hacer tests, además es muy fácil de integrar con las herramientas vistas anteriormente.

Además para hacer la interfaz he usado **Tkinter**, para las conexiones he usado la biblioteca **socket**, para obtener las funciones criptográficas para encriptar y desencriptar usando *AES-128 GCM* 2.5 y la biblioteca **hashlib** para extender las claves y ajustarlas a los bloques usados en AES.

8.2. Estructura de archivos

A continuación voy a explicar la estructura de archivos que se ha seguido en el desarrollo de las aplicaciones cliente y servidor, indagando en lo que estos contienen y explicando las distintas clases y funciones que se han utilizado.

8.2.1. Cliente

Este programa consta de dos clases, diversos métodos y funciones repartidas en los siguientes archivos.

- **src/cliente/biblioteca_cliente.py:** En este archivo se encuentran todas las funciones necesarias para encriptar y desencriptar los mensajes.
  - **SERVER_IP**, variable en la que se almacenará la ip del servidor. Por defecto y como para las pruebas se ha ejecutado en local, tiene el valor **127.0.0.1**.
  - **SERVER_PORT**, variable que almacena el puerto del servidor por el que se comunicará la aplicación, tiene el valor **3333**.
  - **rellenar_bloque(mensaje)**, función que recibe como entrada un mensaje y lo amplia para que tenga un tamaño múltiplo de 16.
  - **vaciar_bloque(mensaje)**, función que recibe como entrada un mensaje y elimina los espacios en blanco añadidos para aumentar su tamaño.
  - **encriptar(mensaje,passw)**, función que recibe como parámetros de entrada un mensaje y una contraseña, y encripta los mensajes

usando *AES-128* con el modo *GCM*. Para poder recibir cualquier contraseña como entrada se amplia la contraseña usando una función resumen. La función devuelve tres valores: el mensaje cifrado, el nonce resultante del cifrado y tag, una etiqueta que se utiliza para asegurarse al desencriptar que el mensaje no ha sido manipulado.

- **desencriptar(mensaje_cifrado, passw, tag, nonce)**, función que recibe un mensaje, una contraseña, una etiqueta y un nonce, y devuelve el mensaje desencriptado.

- **src/cliente/interfaz.py:** En este archivo se encuentran las clases con sus respectivos modos para ejecutar el programa usando Tkinter.

  - **Clase Ventana1**, esta clase cuenta con los métodos **enviar** y **get_mensaje**, estos se usan para enviar y recibir mensajes a través de sockets. Además cuenta con diversos atributos en los cuales se almacenan todos los objetos usados en la interfaz como son: **server_socket** para almacenar el socket para conectarse al servidor, **txt** donde se imprimirán los mensajes y **entrada** por donde se introducirán los mensajes para enviarlos entre otros. Esta es la ventana principal de la aplicación.
  - **Clase Ventana2**, esta clase cuenta con un solo método que servirá para recuperar los datos introducidos por el usuario y enviarlos al servidor, una vez hecho esto elimina la instacia actual y crea una nueva de la clase Ventana2. Además cuenta con diversos atributos necesarios para almacenar la información como son los atributos **emisor** que almacena el nombre de usuario o **receptor** que almacena el usuario con el que se intercambiarán los mensajes.

- **src/cliente/cliente.py:** En este archivo se encuentra el programa principal, crea una instancia de la clase Ventana2 y crea un socket para conectarse con el servidor.

8.2.2. Servidor

Este programa es más sencillo que el programa cliente, ya que como he mencionado anteriormente, no consta de interfaz gráfica. Además, como la aplicación la he diseñado para que tenga un cifrado extremo a extremo no interviene en el encriptado y desencriptado de los mensajes. Consta de solo dos archivos y no tiene ninguna clase.

- **src/servidor/servidor.py:** Es el programa principal y lo que hace es crear un socket por el que se conectaran todos los clientes, una función llamada **escuchar_clientes** que se encarga de cada vez que

recibe un mensaje, lo fragmenta y obtiene el emisor, el receptor, el mensaje encriptado y la etiqueta y el nonce usados en AES-128 con el modo GCM. Una vez fragmentado el mensaje se envia al usuario receptor. A continuación se crea un **Thread** con la función anterior para paralelizar la entrada de mensajes y el envío.

- **src/servidor/biblioteca_servidor.py:** Archivo en el que se almacenan algunas variables globales del programa, las más importantes son: **SERVER_IP** variable que almacena la dirección del servidor, como se ejecuta en local, en este caso **0.0.0.0** y la variable **SERVER_PORT**, variable que almacena el puerto que se va a usar y por defecto vale **3333**.

8.2.3. Mensaje

Para codificar los mensajes he usado un string separando cada bloque con un substring llamado **separador** que vale <->. El mensaje tiene la siguiente estructura.

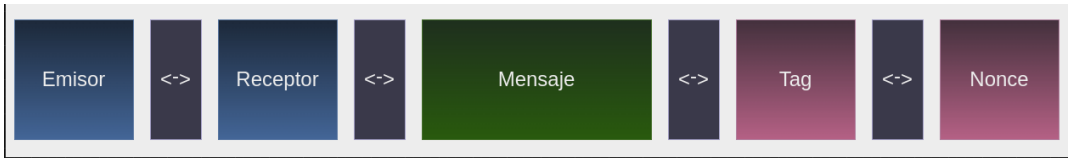


Figura 8.1: Esquema de la codificación del mensaje.

Donde *emisor* es el identificador del usuario que envía los mensajes, *receptor* es el identificador del usuario que los recibe, *mensaje* es el mensaje que se envía, *tag* es la etiqueta usada en AES-128 con el modo GCM para validar los mensajes y *nonce* es un número arbitrario que se usa solo una vez para garantizar la seguridad del mensaje frente ataques *playback*.

8.3. Funcionamiento de la aplicación

Para ejecutar la aplicación, primero se tiene que ejecutar el servidor en algún dispositivo y mientras esté en funcionamiento, ejecutar el cliente.
Para lanzar ambas aplicaciones hay que ejecutar el comando *poe run*. Una vez ejecutado el cliente nos sale la primera interfaz donde introducimos el nombre de usuario con el que nos vamos a identificar y el nombre del usuario con el que nos vamos a comunicar 8.2.

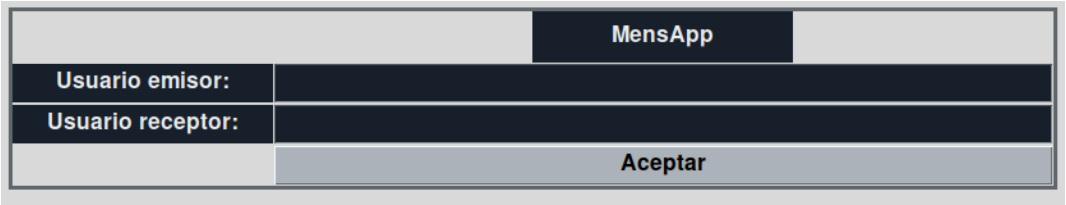


Figura 8.2: Registro en la aplicación.

Una vez registrados entramos en el chat con el usuario receptor. El primer mensaje que se envía, similar a otras aplicaciones de mensajería, no esta cifrado pero servirá para sincronizar ambos clientes y realizar el intercambio de claves *Diffie-Hellman*. Para el intercambio he elegido como número primo 13 y como generador 2. Una vez realizado el intercambio, que se hace de manera automática y pasa desapercibido a los usuarios, se cifran todos los mensajes. En 8.3 podemos ver un ejemplo de conversación entre dos usuarios. El código de la aplicación se encuentra en un repositorio de GitHub.

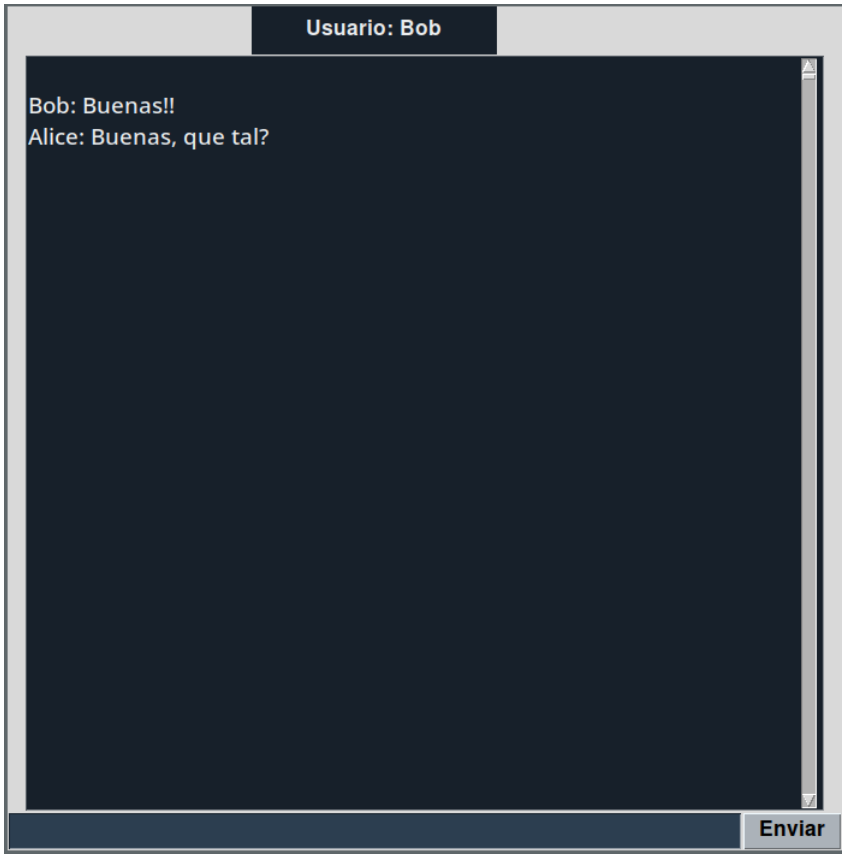


Figura 8.3: Intercambio de mensajes en la aplicación.

Capítulo 9

Conclusiones y trabajos futuros

Bibliografía

[1] AATA Estructura de Cuerpos Finitos. Acceso: 04-07-2023. URL: http://abstract.ups.edu/aata-es/section-finite-field.html.

[2] Block Cipher Mode operation. Acceso: 28-03-2023. URL: https://en.wikipedia.org/wiki/Block{_}cipher{_}mode{_}of{_}operation.

[3] Cómo iMessage envía y recibe mensajes de forma segura - Soporte técnico de Apple (ES). Acceso: 22-06-2023. URL: https://support.apple.com/es-es/guide/security/sec70e68c949/web.

[4] Descripción general de la seguridad de iMessage - Soporte técnico de Apple (ES). Acceso: 22-06-2023. URL: https://support.apple.com/es-es/guide/security/secd9764312f/web.

[5] Dr Clifford Cocks CB — Graduation — University of Bristol. Acceso: 20-03-2023. URL: http://www.bristol.ac.uk/graduation/honorary-degrees/hondeg08/cocks.html.

[6] Federal Information Processing Standards Publication 197 Announcing the ADVANCED ENCRYPTION STANDARD (AES). Acceso: 13-03-2023. URL: http://csrc.nist.gov/csor/.

[7] How WhatsApp enables multi-device capability - Engineering at Meta Acceso: 09-05-2023. URL: https://engineering.fb.com/2021/07/14/security/whatsapp-multi-device/.

[8] Mobile Protocol: Detailed Description. Acceso: 06-03-2023. URL: https://core.telegram.org/mtproto/description.

[9] MTProto Mobile Protocol. Acceso: 24-02-2023. URL: https://core.telegram.org/mtproto/.

[10] NP-intermediate - Wikipedia. Acceso: 16-05-2023. URL: https://en.wikipedia.org/wiki/NP-intermediate.

[11] Servicio de identidad (IDS) de Apple: - Soporte técnico de Apple (ES). Acceso: 22-06-2023. URL: https://support.apple.com/es-es/guide/security/aside/secf752dc2e2/1/web/1.

[12] SHA-1 - Wikipedia. Acceso: 23-05-2023. URL: https://en.wikipedia.org/wiki/SHA-1{#}cite{_}note-20.

[13] SHA-2 - Wikipedia, la enciclopedia libre. Acceso: 23-05-2023. URL: https://es.wikipedia.org/wiki/SHA-2.

[14] Description of SHA-256, SHA-384 AND SHA-512. *ACM Transactions on Programming Languages and Systems*, 9(July):9, 2016. arXiv:43543534534v343453.

[15] WhatsApp Encryption Overview. 2023.

[16] AEPD-EDPS. *Introducción al hash como técnica de seudonimización de datos personales*. 2019.

[17] Eli Biham and Rafi Chen. Near-Collisions of SHA-0. pages 290–305, 2004.

[18] Line Corporation. LINE Encryption Overview.

[19] Morris Dworkin. Recommendation for block cipher modes of operation: methods and techniques. 2001. doi:10.6028/NIST.SP.800-38a.

[20] Adrián Guzmán. Notas del curso de Criptografía. 2008.

[21] Hugo Krawczyk. Cryptographic Extraction and Key Derivation: The HKDF Scheme. *Cryptology ePrint Archive*, 6223, 2010.

[22] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a Shambles * First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust. URL: https://sha-mbles.github.io/.

[23] Francisco Javier Lobillo Borrero. Apuntes de Criptografía. 2019. URL: https://digibug.ugr.es/handle/10481/60106.

[24] Manuel José Lucena López. *Criptografía y Seguridad en Computadores*. Jaén: Escuela Politécnica Superior de España, 2011.

[25] Ángel del Río Mateos. *Introducción a la Criptología*. 2021.

[26] David A Mcgrew and John Viega. The Galois/Counter Mode of Operation (GCM).

[27] Marino Miculan and Nicola Vitacolonna. Automated symbolic verification of Telegram's MTProto 2.0. *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021*, 2021.

[28] Wouter Penard and Tim van Werkhoven. On the secure hash algorithm family. *Cryptography in Context*, pages 1–18, 2008. URL: `https://www.staff.science.uu.nl/{~}tel00101/liter/Books/CrypCont.pdf`.

[29] Adi Shamir and Eran Tromer. On the Cost of Factoring RSA-1024.

[30] Harshvardhan Tiwari. Merkle-Damgård construction method and alternatives: A review. *Journal of Information and Organizational Sciences*, 41(2):283–304, 2017. `doi:10.31341/jios.41.2.9`.