UNIVERSIDAD
DE GRANADA

Criptosistemas en aplicaciones de mensajería

Trabajo de fin de grado Doble grado en Ingeniería Informática y Matemáticas

Autor
Luis Tormo Fabios

Director
Pedro A. García Sánchez

—

Granada, mes de 201

Índice general

Capítulo 1

Introducción:

Tengo que hablar de que va a ir el TFG
Introducir al cifrado extremo a extremo
Historia del cifrado

1.1. Cifrado a extremo a extremo

El cifrado extremo a extremo es un sistema de comunicación en el cual solo pueden leer los mensajes aquellos usuarios que se están comunicando evitando incluso su decodificación por parte de proveedores de telecomunicaciones, proveedores de internet y el propio servicio de comunicación

Capítulo 2

Criptografía y Curvas Elípticas

En este capítulo se introducirá la teoría sobre criptografía y curvas elípticas necesaria para entender la base detrás de los criptosistemas usados en las aplicaciones de mensajería más populares.

2.1. Objetivos de la criptografía

En este apartado voy a hablar sobre los objetivos de un criptosistema y los posibles ataques que se le pueden hacer, así como una introducción a la criptografía simétrica y asimétrica y su uso en las aplicaciones de mensajería que nos ayudará a entender los distintos criptosistemas de los que se hablará después. Mayormente la información de este apartado ha sido obtenida de [19] para los criptosistemas simétricos y [21] para los cifrados asimétricos.
Los principales objetivos que debe cumplir todos los criptosistemas son:

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 a posteriori negar que se realizado tal envío.

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 veamos qué el uso de los criptosistemas simétricos y asimétricos en las aplicaciones de mensajería y posteriormente una introducción más técnica de estos.

2.1.1. Criptosistemas simétricos y asimétricos en las aplicaciones de mensajería

Los criptosistemas simétricos y asimétricos conforman una parte fundamental de las aplicaciones de mensajería, ambos son usados a la par de manera complementaria.
Los criptosistemas simétricos debido a su velocidad de cifrado, su uso reducido de recursos y su mejor manejo de grandes cantidades de datos se suelen utilizar para cifrar los mensajes. Pero como 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 cifrados 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.

2.1.2. Criptosistema simétrico

Un criptosistema simétrico es un criptosistema en el cual se utiliza una sola clave para cifrar y descifrar 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.
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:

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

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

2.1.3. 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 conozcla la clave privada.
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.

Un criptosistema asimétrico viene definido por dos aplicaciones:

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

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

tales que para cualquier clave privada $k' \in \mathcal{K}'$ obtenemos la clave pública como $P(k') = k$ 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 que sea imposible o el periodo sea muy largo en caso de solo conocerse k.

2.1.4. Cifrados de bloque

A continuación se introducirá los cifrados de bloque, necesarios para entender el algoritmo **Rindael AES** que se explicará posteriormente. La información ha sido obtenida de [19].
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 [15].
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 $i \in \{1, \dots, l\}$ hacer
    $m_{[i]} = D_k(c_{[i]})$
    Devolvemos $m_{[1]} \dots m_{[l]}$

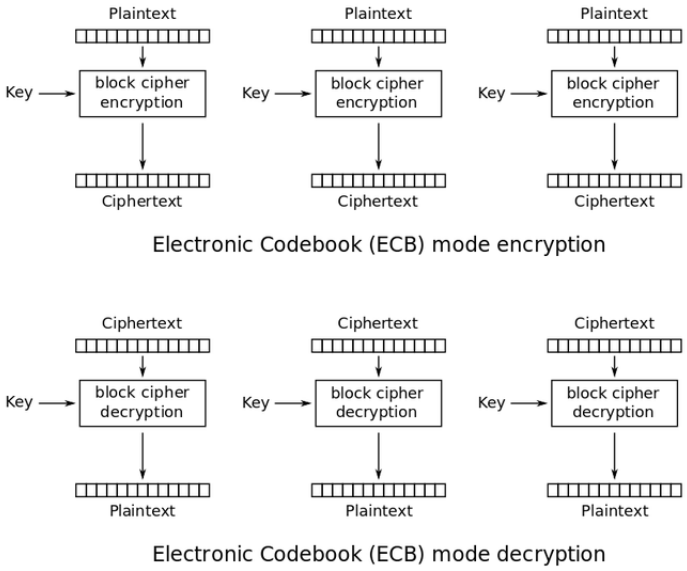


Figura 2.1: Esquema del cifrado y descifrado del modo ECB [1].

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

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

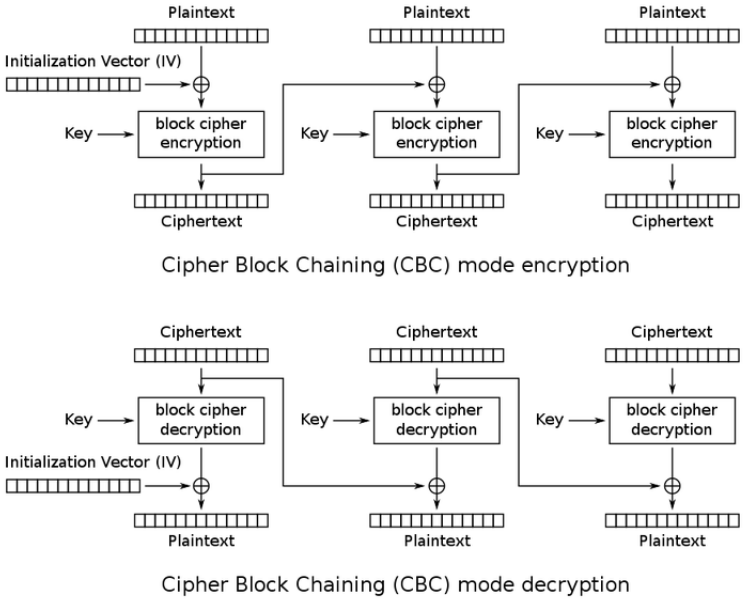


Figura 2.2: Esquema del cifrado y descifrado del modo CBC [1].

- **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 $m_{[1]} \dots m_{[l]}$

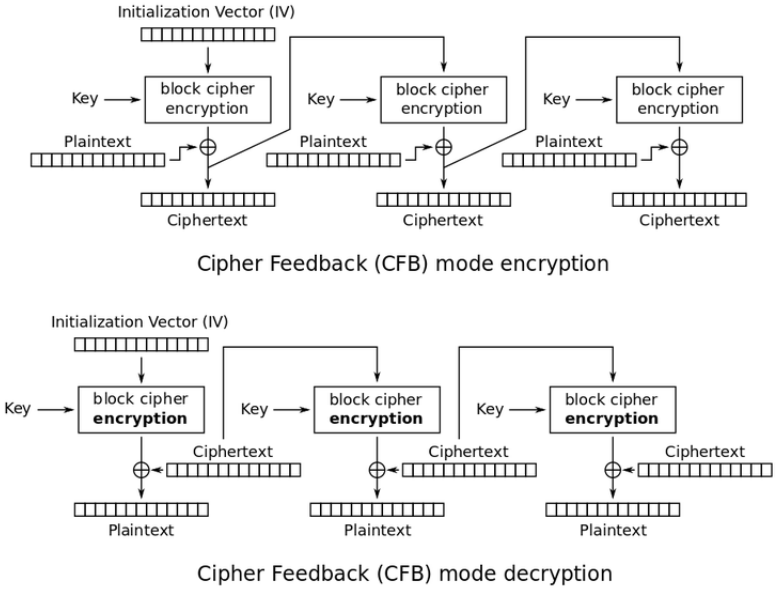


Figura 2.3: Esquema del cifrado y descifrado del modo CFB [1].

- **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 $m_{[1]} \dots m_{[l]}$

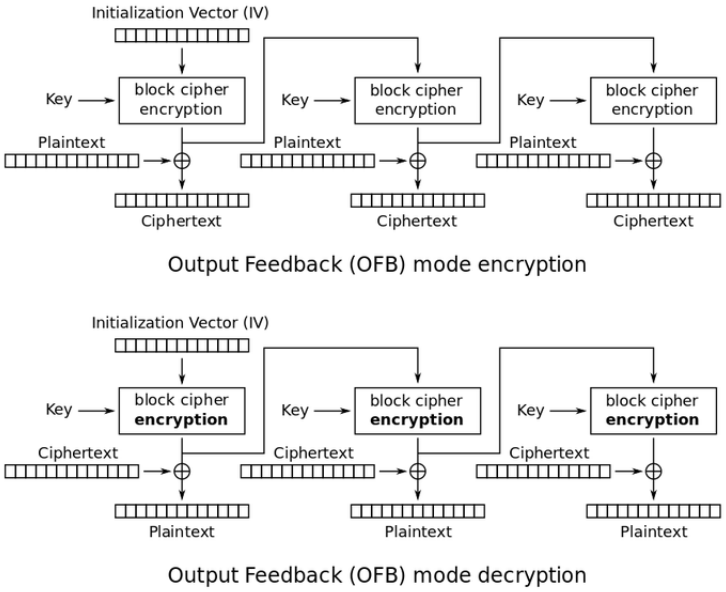


Figura 2.4: Esquema del cifrado y descifrado del modo OFB [1].

Como podemos ver, el más sencillo es ECB ya que lo único que hace es fragmentar el mensaje en bloques y encriptar individualmente cada bloque. En CBC, CFB y OFB se parte de un bloque inicial y se generan bloques nuevos de manera recursiva operando con ellos de manera disinta en fución de cada modo.
En CBC se realiza la operación $\oplus$ de cada bloque generado encriptandose 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 de resultado 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 $||$.
Y en OFB se realiza la operación $\oplus$ de el resultado de encriptar el bloque generado previo con un bloque 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.

2.2. El algoritmo Rijndael AES

En esta sección hablaré sobre el cifrado Rijndael AES el cual es un cifrado de bloque simétrico muy utilizado actualmente por aplicaciones como *Telegram*, *WhatsApp* y *FacebookChat* entre otras.
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 [3] 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 [20] y de [**?**].

2.2.1. Estructura de AES

AES es un algoritmo que se basa en aplicar un número determinado de rodas 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. El bloque a cifrar o descifrar se traslada directamente byte a byte sobre la matriz de estado de columna en columna$(a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}...)$

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

Cuadro 2.1: Ejemplo de matriz de estado con $N_b = 4$(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

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

Cuadro 2.2: Ejemplo de clave con $N_k = 4$(128 bits).

algoritmo AES con n rondas quedaría:

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 roda 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 2.3: Número de rodas 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* y 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 *or-exclusivo* entre el estado intermedio y la subclave correspondiente a cada ronda.

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

Antes de desarrollar las rondas de AES y posteriormente la teoría de Curvas Elípticas en $\mathrm{GF}(2^n)$ introduciré el cuerpo $\mathrm{GF}(2^n)$ el cual tiene una serie de propiedades que lo hacen muy interesante y justifican su uso tan extendido en criptografía.
El conjunto $\mathbb{Z}_2[x]$ es el conjunto de polinomios con coeficientes en $\mathbb{Z}_2$ es decir, el el conjunto de polinomios cuyos coeficientes solo valen 0 o 1. Por lo que 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.
Escogiendo un polinomio irreducible en $\mathbb{Z}_2$ podemos generar un cuerpo de Galois. Este conjunto es representado como $\mathrm{GF}(2^n)$, donde n es el grado del polinomio irreducible que lo genera.
Las principales ventajas que tiene trabajar con $\mathrm{GF}(2^n)$ es que permite llevar a cabo implementaciones de una sencillez significativa respecto a la de los demás. Por lo que teniendo el mismo orden de complejidad, se multiplica significativamente la velocidad y además permite simplificar el diseño de los circuitos. Esto último hace que se obtengan sistemas con mejores prestaciones y mejor precio.
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 [16].
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 se define el cuerpo $\mathbb{F}_{p^n}$ al único cuerpo existente con p^n elementos. En particular para trabajar con $\mathbb{F}_{256}$ $p = 2$ 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 done 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 $x^8 + x^6 + x^3 + x^2 + 1$ entre otros.
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 $\mathbb{Z}_{2_{x^8+x^4+x^3+x+1}}$.
Los elementos que conformaran al 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 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 computacional muy elevada por lo que tendremos que buscar algún método basado en los resultados de cuerpos finitos para intentar disminuirla.

Definición 2.1 *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.*

Por lo que 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 independientes y distintos de cero, por lo que 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 $\alpha^n = \alpha^{n \mod q-1}$.

Teorema 2.2 *Todo cuerpo finito tiene al menos un elemento primitivo.*

Salvo para $q = 2$ se tiene que 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 $\mathbb{F}_{\not\in\not\in\not\in}$ 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 se va a seguir para 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.

	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 2.4: Tabla de los antilogaritmos de $[x+1]$

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		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 2.5: 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:

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

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 $D6 \cdot 75 = F8$.

2.2.3. Las Rondas de AES

Una vez visto algunas propiedades de los cuerpos finitos vamos a hablar de las operaciones que se realizan en 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 2.3.
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 2.6.
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 2.6: Valores de c_i según el tamaño de bloque N_b

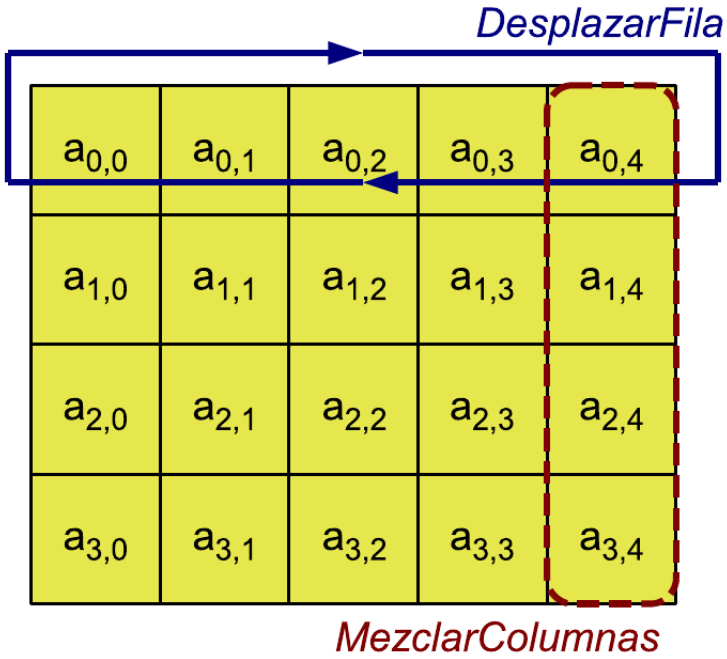


Figura 2.5: Esquema de las funciones *MezclarColumnas* y *DesplazarFila* [20]

MezclarColumnas

Durante la aplicación de esta función se considera cada columna del vector de estado se considera un polinomio cuyos coeficientes pertenecen a GF(2^8) y se multiplica módulo $x^4 + 1$ por: $c(x) = 03x^3 + 01x^2 + 01x + 02$ donde 03 es el valor hexadecimal que se obtiene concatenado los coeficientes binarios del polinomio correspondiente en GF(2^8), en este caso sería 00000011 y por tanto $x + 1$ análogamente se haría con

los demás.
La inversa de $MezclarColumnas$ se obtiene multiplicando cada columna de la matriz de estado por el polinomio: $d(x) = 0Bx^3 + 0Dx^2 + 09x + 0E$.

2.2.4. 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 $N_k <= 6$:

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 $i\ mod N_k = 0$
        $tmp \leftarrow Sub(Rot(tmp)) \oplus Rc(i/N_k)$
    $W(i) \leftarrow W(i-N_k) \oplus tmp$

- 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 $i\ mod N_k = 0$
        $tmp \leftarrow Sub(Rot(tmp)) \oplus Rc(i/N_k)$
    Si $i\ mod N_k = 4$
        $tmp \leftarrow Sub(tmp)$
    $W(i) \leftarrow W(i-N_k) \oplus tmp$

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 $x^8 + x^4 + x^3 + x + 1$.

2.3. Criptosistema de Rivest-Shamir-Adleman, RSA

En esta sección hablaré sobre el cifrado RSA y su funcionamiento, cifrado que como AES, está muy extendido y es utilizado por muchas aplicaciones para cifrar y validar los mensajes.
RSA es llamado así en honor a sus creadores Ron Rivest, Adi Shamir y Loenard Adleman, fué 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[2].
Este criptosistema está basado en el *Teorema de Euler* y en particular en la *Proposición 2.2.*

Teorema 2.3 *(Teorema pequeño de Fermat) Sea $a \in \mathbb{Z}$ y p un número primo tal que* $\mathrm{mcd}(a, p) = 1$. *Entonces*

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

Demostración
Sea $a \in \mathbb{Z}$. Tomamos los $p-1$ primeros múltiplos positivos de a que serán de la forma $a, 2a, \ldots, (p-1)a$. El resto resultante de dividir los $p-1$ múltiplos positivos de a por p corresponden a 1,2,3,$\ldots, p-1$.
Multiplicando ahora todas las congruencias obtenemos

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

Como tenemos que $p \nmid (p-1)!$ se cumple que $\mathrm{mcd}(p, p-1) = 1$ y por tanto cancelando en la expresión anterior obtenemos

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

□

Teorema 2.4 *(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

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

□

Donde $\phi(n)$ es la *función de Euler* definida como $\phi(n) = |\mathbb{Z}_n^*|$ y se puede calcular de la siguiente forma

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

Teorema 2.5 *(Teorema Chino del Resto) Sean $a_i \in \mathbb{Z}$ y $p, q \in \mathbb{N}$ tales que $\text{mcd}(p, q)$ con $i = 1, 2$. 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

$$x \equiv a_1 \cdot c_1 \cdot d_1 + a_2 \cdot c_2 \cdot d_2 \mod n,$$

donde se cumple

$$c_1 = \frac{n}{p},$$

$$c_2 = \frac{n}{q},$$

y

$$c_1 \cdot d_1 \equiv 1 \mod p,$$

$$c_2 \cdot d_2 \equiv 1 \mod q.$$

Proposición 2.6 *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, $mcm(a, n) = 1$. 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 $a^x \equiv a \mod n$. □

2.3.1. Descripción de RSA

El contenido de esta sección se basa en [21]. El funcionamiento de RSA es el siguiente.
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 [24] 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:

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

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

2.3.2. Ataques

Como hemos visto anteriormente RSA puede ser vulnerable en función de los números primos que se elijan y el tamaño de estos. En este apartado veremos algunos posibles ataques que se podrían llevar a cabo [19].

Ataque por módulo común.

Este ataque se da cuando hay una mala elección de las claves, típicamente cuando se tienen que generar múltiples claves para varios usuarios y para

optimizar, se utiliza el mismo módulo n para todos.
Supongamos que tenemos dos claves públicas con el mismo módulo, (n, e_1) y (n, e_2) y además ser verifica que $\text{mcd}(e_1, e_2) = 1$. Supongamos que $re_1 + se_2 = 1$ con $r<0<s$. Como se ha visto el mensaje cifrado respectivamente será:

$$\begin{aligned} c_1 &= m^{e_1} \mod n, \\ c_2 &= m^{e_2} \mod n. \end{aligned}$$

Si $(c_1, n) \neq 1$, podemos factorizar n y romper la clave, por lo que podemos suponer que $c_1 \in \mathcal{U}(\mathbb{Z}_n)$. Calculamos c_1^{-1} con el algoritmo extendido de Euclídes tenemos que:

$$(c^{-1})^{-r} c_2^s \equiv (m^{e_1})^r (m^{e_2})^s \equiv m^{e_1 r + e_2 s} = m \mod n.$$

Ataque por exponente pequeño.

Este ataque se puede hacer cuando se elige un primo muy pequeño y se cifra el mismo mensaje con distinto módulo. En este caso se puede recuperar el mensaje original aplicando el Teorema Chino del Resto. Supongamos a varios receptores con claves públicas (n_i, e), con $1 \leq i \leq r$, tal que $\text{mcd}(n_i, n_j) = 1$ si $i \neq j$ ya que en caso contrario, se podría factorizar el módulo correspondiente mediante el cálculo del MCD.
Llamamos $c_i = m^e \mod n_i$ para cada $1 \leq i \leq r$. Seleccionamos $\{i_1, ..., i_e\} \subseteq \{1, ..., r\}$ y empleando el inverso del isomorfismo de anillos

$$\chi : \mathbb{Z}_{n_{i_1} \dots n_{i_e}} \to \mathbb{Z}_{n_{i_1}} \times \cdots \times \mathbb{Z}_{n_{i_e}}$$

obtenido gracias al Teorema Chino del Resto, se puede calcular

$$\chi^{-1}(c_{i_1}, \dots, c_{i_e}) = m^e \mod n_{i_1}, \dots, n_{i_e}.$$

Dado $m^e < n_{i_1} \dots n_{i_e}$, se puede obtener m calculando la raíz e-ésima en $\mathbb{Z} \subseteq \mathbb{R}$, siempre y cuando e no sea muy grande.

Ataque con primos muy próximos.

Este ataque se da cuando se eligen dos primos muy próximos entre sí. Dado $n = pq$, con $p<q$, tenemos que

$$n = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2.$$

Si p y q son cercanos, $s = \frac{p-q}{2}$ es pequeño y $t = \frac{p+q}{2}$ es un entero ligeramente mayor que $\sqrt{n}$ tal que $t^2 - n^2$ es un cuadrado perfecto. Probando sucesivamente con valores mayores que $\sqrt{n}$ hasta encontrar una descomposición $n = t^2 - s^2$, tenemos que $p = t + s$ y $q = t - s$.

2.3.3. 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:

$$C_B(S) = S \mod n_B.$$

   y se lo envía a B.

3. B recibe $C_B(S) = S^{e_B}$ 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.

2.4. 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 ha sido obtenida de [21] y la de *Diffie-Hellman* ha sido obtenida de [20].
El Problema del Logaritmo Discreto es definido de la siguiente forma:

Definición 2.7 *Sea S un semigrupo finito. El Problema del Logaritmo Discreto en el semigrupo S es el de resolver ecuaciones 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 por ahora [8].
Una vez visto el problema de logaritmo discreto, se explicará el intercambio de claves *Diffie-Hellman*.

2.4.1. 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 2.8 *Dado el conjunto $\mathbb{Z}_{p'}^*$ con p primo, diremos que $\alpha \in \mathbb{Z}_p^*$ es un generador de $\mathbb{Z}_{p'}^*$ si se cumple:*

$$\forall b \in \mathbb{Z}_{p'}^*,\ \exists i \ tal \ que \ \alpha^i = b$$

Definición 2.9 *(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* α^{ab} mód 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 $\alpha de \mathrm{Z}_{p'}^*$ 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
$$\alpha^x \quad \text{mód } p$$
2. Análogamente B escoge un número aleatorio y, comprendido entre 1 y $p-2$ y envía a A el valor
$$\alpha^y \quad \text{mód } p$$
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 solamente A y B acaban conociendo el valor de K.
A continuación se introducirá la teoría de curvas Elípticas ya que nos permitirá redefinir el intercambio de claves usando estas, generando un problema mucho más complejo y con mayor seguridad el cual es el que se utiliza en aplicaciones de mensajería como **WhatsApp** y **Telegram**.

2.5. Curvas Elípticas en Criptografía

La criptografía en curvas Elípticas es considerada como uno de los campos de las matemáticas con más futuro en la criptografía asimétrica. Esto es debido a sus propiedades que dan lugar a problemas de una gran complejidad computacional análogos a los que presenta la aritmética modular. Esto permite que sean utilizadas en algunos algoritmos asimétricos como

puede ser el intercambio de claves *Diffie-Hellman* que se verá más adelante. A priori 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 [20].

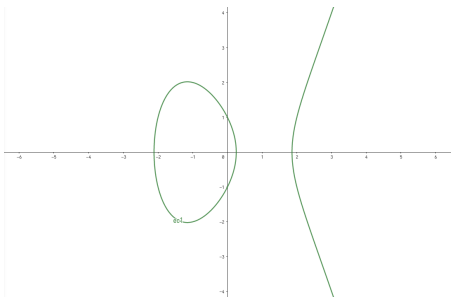
2.5.1. Curvas Elípticas en $\mathbb{R}$

Definición 2.10 *Una curva definida en $\mathbb{R}$ es el conjunto de puntos del plano (x,y) que cumplen la siguiente ecuación:*

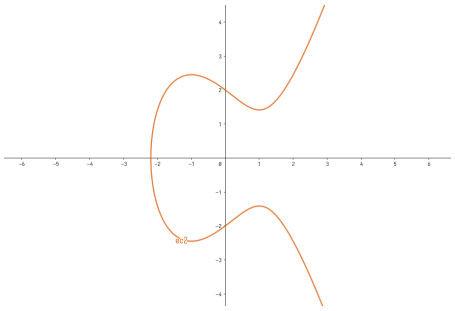
$$y^2 = x^3 + ax + b.$$

Donde los coeficientes a,b $\in \mathbb{R}$ defininen de manera unívoca a la curva.

Algunas curvas en $\mathbb{R}$ son:



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



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

Si se verifica que $x^3 + ax + b$ no tiene raíces múltiples, tendremos que la curva junto con el punto $\mathcal{O}$ llamado *punto en el infinito* y la operación $+$ que se definirá a continuación es lo que se denomina el *grupo de curva elíptica E(R).*
El punto $\mathcal{O}$ es un punto imaginario situado a una distancia infinita que no tendrá ningún valor en particular.

Operación + en $\mathbb{R}$

Definido el conjunto en el que se trabajará, definiremos una ley de composición interna $+$.
Sean los puntos $r = (r_x, r_y)$, $s = (s_x, s_y)$, $p = (p_x, p_y)$, $t = (t_x, t_y)$ con $r, s, p, t \in E(\mathbb{R})$ la operación $+$ se define como sigue:

- $r + \mathcal{O} = \mathcal{O} + r = r, \ \forall r \in E(\mathbb{R})$.

- Si $r_x = s_x$ y $r_y = -s_y$, entonces $r = -s$ y además $r + s = s + r = \mathcal{O}$.

- Si $r \neq s$ y $r \neq -s$, $r + s = p$ donde p será el opuesto del punto que corta la recta que une r y t con la curva.

- Para sumar un punto p con sigo mismo si $p_y \neq 0$ se usa la tangente de la curva en p. Luego tendremos que $t = p + p$ sera el opuesto de ese punto. Si $p_y = 0$ entonces la tangente de la curva será perpendicular al eje de abcisas, por lo que se podría considerar que corta la curva en el infinito, luego $p + p = \mathcal{O}$.

- Para sumar n veces un punto p tenemos que si $p_y \neq 0$ entonces sumar n veces p será equivalente a multiplicar p por el escalar n y se representará como np. Si $p_y = 0$ entonces la suma será:

$$\begin{gathered} 2r = r + r = \mathcal{O}, \\ 3r = 2r + r = \mathcal{O} + r = r, \\ 4r = 3r + r = r + r = \mathcal{O}, \\ \dots \end{gathered}$$

La suma de curvas elípticas de manera algebraica se define de la siguiente forma:
Dados $r = (r_x, r_y)$ y $s = (s_x, s_y)$, tal que $r - s$, tenemos que $r + s = t$ donde $d = \frac{r_y - s_y}{r_x - s_x}$, $t_x = d^2 - r_x - s_x$, $t_y = -r_y + d(r_x - t_x)$. De manera gráfica quedaría de la siguiente manera:

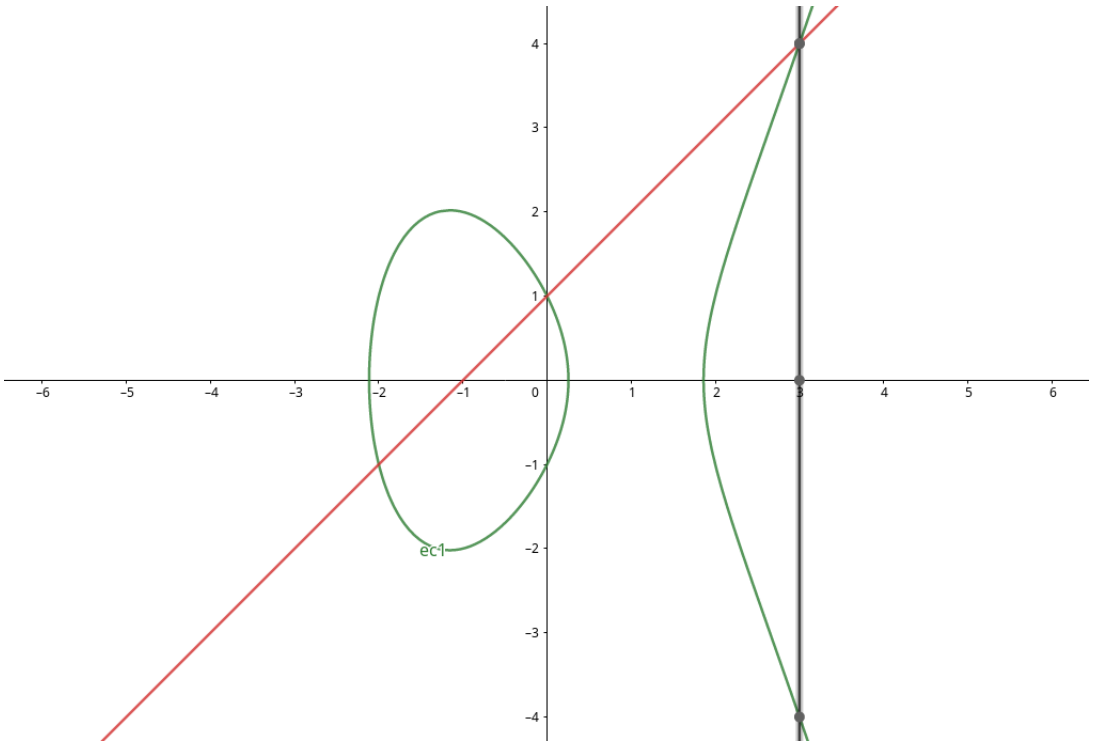


Figura 2.6: Suma en la curva $y^2 = x^3 - 4x + 1$ con $r = (-2, -1)$, $s = (0, 1)$, $-t = (3, 4)$ y $t = (3, -4)$

Una vez definidas las curvas elípticas en $\mathbb{R}$ definiremos las curvas en los cuerpos de Galois GF(n) y GF(2^n) cuyo uso esta muy extendido en la criptografía para redefinir el problema de logaritmo discreto.

2.5.2. Curvas Elípticas en GF(n)

Un cuerpo de Galois GF(n) es el grupo finito generado por un número primo n. En este conjunto todos los elementos tienen un inverso, menos el cero por lo que están permitidas las operaciones de suma, resta, multiplicación y división.
De manera natural se define el conjunto $E(\text{GF}(n))$ como los puntos (x, y) que verifican la siguiente ecuación:

$$y^2 \equiv x^3 + ax + b \quad \text{mód } n$$

Donde al igual que en $E(\mathbb{R})$, $a, b \in 1..n$ definen de manera unívoca una curva elíptica.

Operación + en GF(n)

Sean los puntos $r = (r_x, r_y)$, $s = (s_x, s_y)$, $p = (p_x, p_y)$, $t = (t_x, t_y) \in E(\mathrm{GF}(n))$, definimos la operación + como sigue:

- $r + \mathcal{O} = \mathcal{O} + r = r$, $\forall r \in \mathrm{GF}(n)$.
- Si $r_x = s_x$ y $r_y = s_x + s_y$, entoces se dice que r es el opuesto de s, se nota como $r = -s$ y se verifica que $r + s = s + r = \mathcal{O}$.
- Si $r \neq s$ y $r \neq -s$, $t = r + s$ se calcula como $d = \frac{s_y - r_y}{s_x - r_x} \mod n$, $t_x = d^2 - r_x - s_x \mod n$, $t_y = -r_y + d(r_x - t_x) \mod n$
- Si $p_x = 0$ entonces $2p = \mathcal{O}$

2.5.3. Curvas Elípticas en GF(2^n)

De manera análoga a $E(\mathrm{GF}(n))$ definimos el conjunto $E(\mathrm{GF}(2^n))$ con la diferencia debida a la estructura de GF(2^n), la ecuación de curva elíptica es diferente.
Dado un polinomio irreducible $p(x)$ de grado n, las curvas elípticas se definen como los puntos (x, y) que cumplen la ecuación:

$$y^2 + xy \equiv x^3 + ax^2 + b \mod (p(x))$$

y para que se genere un grupo se tiene que verificar que $b \neq 0$. Los puntos de una curva serán pares de polinomios de grado $n - 1$ y como hemos visto en el apartado anterior, podrán ser representados como cadenas de bits.

Operación + en $E(\mathrm{GF}(2^n))$

Sean los puntos $r = (r_x, r_y)$, $s = (s_x, s_y)$, $p = (p_x, p_y)$, $t = (t_x, t_y) \in E(\mathrm{GF}(n))$, definimos la operación + como sigue:

- $r + \mathcal{O} = \mathcal{O} + r = r$, $\forall r \in E(\mathrm{GF}(2^n))$
- Si $r_x = s_x$ y $r_y = s_x + s_y$, entoces se dice que r es el opuesto de s, se nota como $r = -s$ y se verifica que $r + s = s + r = \mathcal{O}$.
- Si $r \neq s$ y $r \neq -s$, $t = r + s$ se calcula como $d = \frac{s_y - r_y}{s_x - r_x}$, $t_x = d^2 + d + r_x + s_x + a$, $t_y = d(r_x + t_x) + t_x + r_y$.
- Para $t = 2p$ con $p_x \neq 0$ se calcula como $d = p_x + \frac{p_y}{p_x}$, $t_x = d^2 + d + a$, $t_y = p_x^2 + (d + 1)t_x$.
- Si $p_x = 0$ tenemos que $2p = \mathcal{O}$

2.6. El problema del logaritmo discreto usando curvas elípticas. *Diffie-Hellman*

En esta sección se hablará sobre el análogo del problema del logaritmo discreto en curvas elípticas y como resultado un análogo del intercambio de claves *Diffie-Hellman*.

2.6.1. 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 númer $k \in \mathbb{Z}$ 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.

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

Una vez visto el problema del logaritmo discreto en curvas elípticas, se explicará el intercambio de claves $Diffie-Hellman$ usando curvas elípticas. Para ello se explicará previamente la conjetura *Diffie-Hellman*. La información de este apartado la he obtenido de [19].
Fijamos una curva elíptica $E = E(a, b)$ tal que $|E| = hn$ con n primo y h pequeño. Se fija también q un elemento de orden n.

Definición 2.11 *(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:

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 $a \in (2, ..., n-1)$ y le envía a B $p_a = aq$.
- B elige aleatoriamente un número $b \in (2, ..., n-1)$ y le envía a A $p_b = bq$.

- A calcula $a(p_b)$.
- B calcula $b(p_a)$.
- La clave compartida es $(ab)q = a(p_b) = b(p_a)$

2.7. Funciones Hash

Una función resumen o función hash es un proceso 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, la información para esta sección la he obtenido [13].
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: *MD5, 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

2.7.1. MD5

MD5 fue diseñada en 1992 por Ron Rivest como una mejora de la función MD4. Es una de las funciones más usadas hasta la fecha aunque su uso esta disminuyendo debido a que se han encontrado algunas debilidades en esta. Uno de los motivos por los que es tan importante es que sirvió como base para desarrollar las funciones SHA-0, SHA-1 y la familia de funciones SHA-2. La información ha sido obtenida de [25].
En MD5 el mensaje inicial se fragmenta en bloques de 512 bits y la salida es un hash de 128 bits, el proceso de generación de este es el siguiente:

1. 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:

   $$h_j = compresión(M_{j-1}, h_{j-1}).$$

4. h_n es la salida de la función hash.

Donde la función hash funciona de la siguiente manera:

1. Se divide el bloque M_j de 512 bits en bloques 16 bloques de 32 bits $m_0, m_1, ..., m_{15}$.

2. Divide h_{j-1} en 4 registros A, B, C y D como:

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

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

$$A_{i+1} = B_i + ((A_i + \Phi_i(B_i, C_i, D_i) + W_i + T_i) \lll S_i),$$

$$D_{i+1} = A_{i+1} + ((D_i + \Phi_{i+1}(A_{i+1}, B_i, C_i) + W_{i+1} + T_{i+1}) \lll S_{i+1}),$$

$$C_{i+1} = D_{i+1} + ((C_i + \Phi_{i+2}(D_{i+1}, A_{i+1}, B_i) + W_{i+2} + T_{i+2}) \lll S_{i+2}),$$

$$B_{i+1} = C_{i+1} + ((B_i + \Phi_{i+3}(C_{i+1}, D_{i+1}, A_{i+1}) + W_{i+3} + T_{i+3}) \lll S_{i+3}).$$

Donde la operación + es la operación ADD mód 32, T_{i+j} y S_{i+j} ($j = 0, 1, 2, 3$) son constantes dependientes de la iteración y W_i son palabras del mensaje.
En cada ronda se utiliza una función $\Phi_i(X, Y, Z)$ que depende de la iteración:

$$\Phi_i(X, Y, Z) = (X \wedge Y) \vee (\overline{X} \wedge Z),\ 0 \leq i \leq 15,$$
$$\Phi_i(X, Y, Z) = (X \wedge Z) \vee (Y \wedge \overline{Z}),\ 16 \leq i \leq 31,$$
$$\Phi_i(X, Y, Z) = X \oplus Y \oplus Z,\ 32 \leq i \leq 47,$$
$$\Phi_i(X, Y, Z) = Y \oplus (X \vee \overline{(Z)}),\ 48 \leq i \leq 63$$

En la imagen siguiente se puede ver un esquema del proceso de generación del hash usando MD5.

2.7.2. SHA-0

SHA-0 es una función hash que apareció publicado en el Federal Information Processing Standard (FIPS-180) por el NIST en 1993 [23]. Está basado en *MD4* y *MD5*. El algoritmo transforma un mensaje de cualquier tamaño hasta 2^{64} bits y los transforma en hashes de 160 bits.
El funcionamiento de SHA-0 es el siguiente [14]:

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:

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

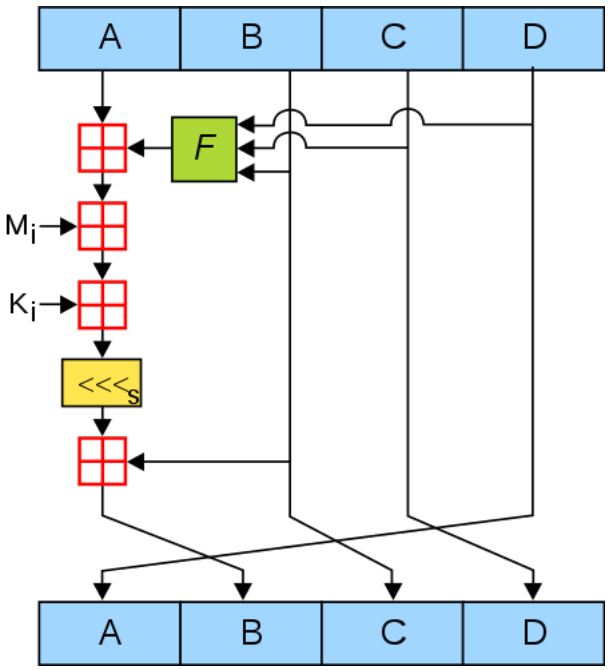


Figura 2.7: Esquema de los pasos seguidos en MD5 [5].

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 2.7.

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 2.7: Funciones y constantes usadas en la función de compresión de SHA-0 [14].

2.7.3. SHA-1

La función SHA-1 es una función hash diseñada en 1995 por la *National Security Agency* (NSA) dado que que se encontró varias colisiones y vulnerabilidades en la función SHA-0 [23].
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 2.8 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 2.8: Funciones y constantes usadas en la función de compresión de SHA-1 [18].

En siguiente imagen 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.

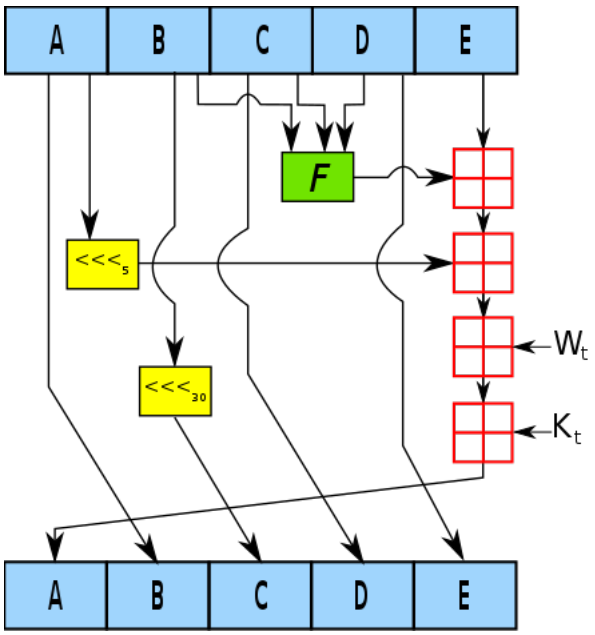


Figura 2.8: Esquema de los pasos seguidos en SHA-0 y SHA-1 [9].

2.7.4. 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 [23]. 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[11]:

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:

$$h_j = \textit{compresión}(M_j, h_{j-1}).$$

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 2.7.

5. La salida de la función sería:

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

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

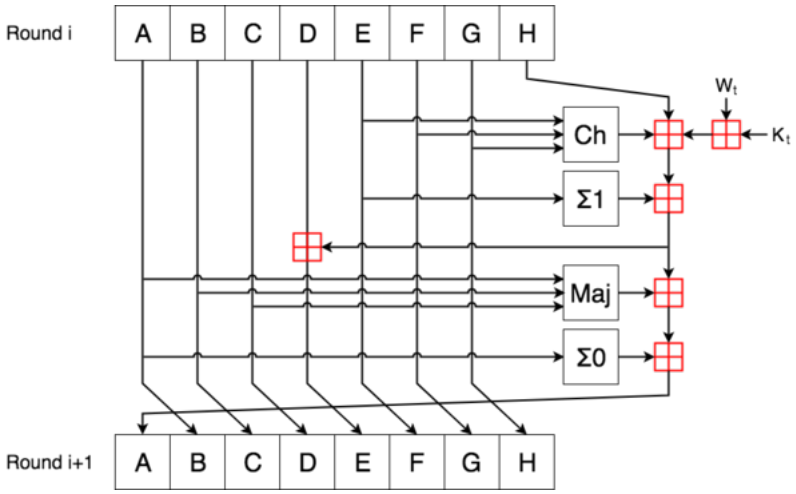


Figura 2.9: Esquema de los pasos seguidos en las funciones de la familia SHA-2 [10].

Con esto concluye el capítulo en el cual se han introducido todas las herramientas necesarias para entender los criptosistemas de las aplicaciones de mensajería. En el próximo capítulo procederé a explicar los criptosistemas utilizados en algunas de las aplicaciones de mensajería más populares.

Capítulo 3

Aplicaciones de Mensajería

En este capítulo explicaré los criptosistemas utilizados por las aplicaciones de mensajería más populares actualmente, estas serán: *Telegram*, *WhatsApp*, *Facebook*, *Signal* y *iMessage*.

3.1. 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* y en 2017 fue reemplazado ya que se encontró una vulnerabilidad en la intercepción de mensajes debido a *SHA-1*.
- **MTProto v2:** En 2017 se actualizó MTProto, en esta versión se cambió el cifrado *SHA-1* por *SHA-256* que es el que se utiliza actualmente y el que se desarrollará a continuación.

La información técnica sobre MTProto que se desarrollará a continuación ha sido obtenida de [22] y [7].

3.1.1. 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 para transmitir los mensajes sobre otro protocolo de red como HTTP, HTTPS, WS, WSS, TCP o UDP.

Y se pueden resumir en:

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:

- **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 o 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 *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 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 *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*.

3.1.2. 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 [6].

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 como 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 clave se está generando. La aplicación tiene una clave publica 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 que 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 y 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 por una petición del servidor. Una vez generado el nuevo salt todos los mensajes tienen que tenerlo

aunque se aceptan los mensajes con el salt previo. 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 sin sesión. Los *Message Identifiers* son divisibles por 4, 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 requiere un reconocimiento explícito. 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 bit 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*. Cabe a destacar que como un contenedor se genera después de su contenido, 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.

Encrypted Message

auth_key_id int64	msg_key int128	encrypted_data bytes

Encrypted Message: *encrypted_data*

Contains the cypher text for the following data:

salt int64	session_id int64	message_id int64	seq_no int32	message_data_length int32	message_data bytes	padding12..1024 bytes

Unencrypted Message

auth_key_id = 0 int64	message_id int64	message_data_length int32	message_data bytes

MTProto 2.0 uses 12..1024 padding bytes, instead of the 0..15 used in MTProto 1.0

3.1.3. 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* ya que MTProto requiere que los tipos de datos estén en formato binario y *TL Language* que sirve para describir el sistema utilizado de tipos y funciones.
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* aunque si pertenecen al *SHA-1* los bytes no son reorganizado.
Una vez introducido 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.

2. El servidor le responde enviándole:

- *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* de claves RSA públicas.

3. El cliente descompone *pq* en factores primos tal que $p<q$. Con esto empieza el intercambio de claves Diffie-Hellman.

4. El cliente envía una nueva consulta que contiene:

   - *nonce*
   - *server_nonce*
   - *p*: Factor obtenido en el paso anterior, es de tipo *long*.
   - *q*: El otro factor obtenido en el paso anterior, al igual que *p* es de tipo *long*.
   - *public_key_fingerprint*: Una de las *fingerprints* obtenida de la lista enviada por el servidor en el paso anterior, es del tipo *long*.
   - *encrypted_data*: Mensaje cifrado obtenido aplicando RSA a *data* y *server_public _key* donde:
     - *data*: Es una serialización de *pq*, *p*, *q*, *nonce*, *server_nonce*, *new_nonce* (un nuevo número aleatorio generado por el cliente y desde este paso conocido por el cliente y el servidor) y *dc* o una serialización de *pq*, *p*, *q*, *nonce*, *server_nonce*, *new_nonce*, *dc* y *expires_in*.
     - *dc* es un identificador de la consulta, es del tipo *int*.
     - *expires_in* es el tiempo en el que expira la consulta, es del tipo *int*.

   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 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*.
  - *answer_with_hash*: Es una generación con la función hash *HASH1* quedando de la siguiente forma: *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*.
  - *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 el cual es un primo usado en Diffie-Hellman, 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 $\frac{p-1}{2}$.
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 de *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.

3.1.4. 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 consiste en:

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 3.1.

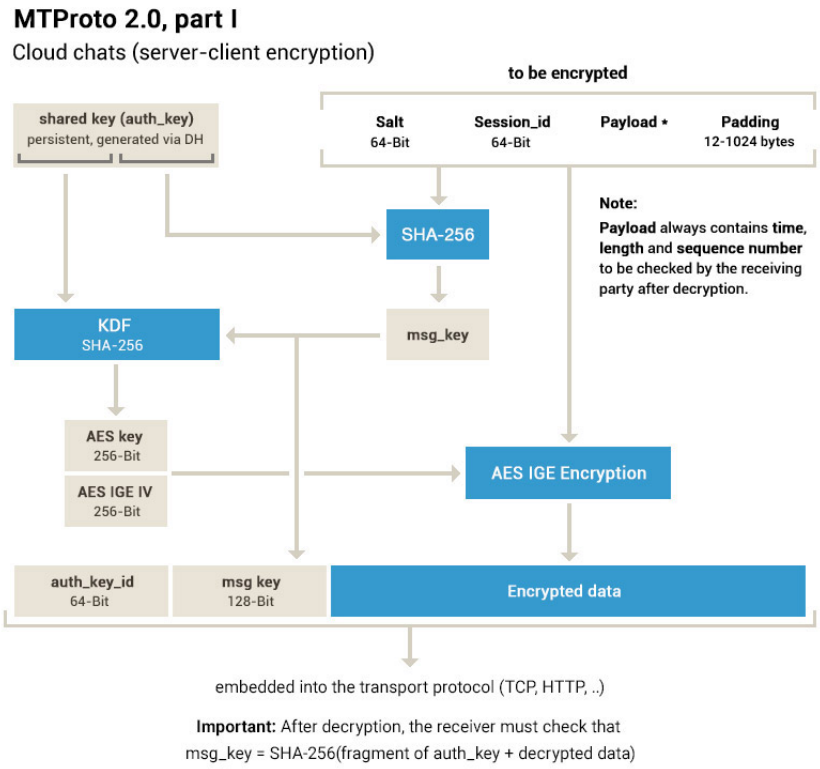


Figura 3.1: Esquema del cifrado de mensajes usado en MTProto 2.0 [7]

3.1.5. 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:

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

- *UDP*

3.2. 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 las cuales son:

- **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 se va a desarrollar a continuación ha sido obtenida de [12].

3.2.1. 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:

- **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*.

3.2.2. 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.

3.2.3. 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.

3.2.4. 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.

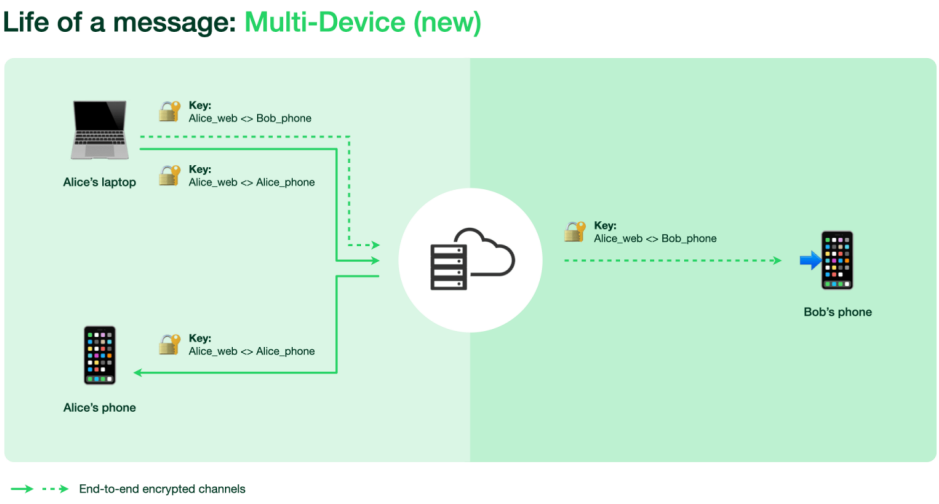


Figura 3.2: Diagrama de un conexión entre dos dispositivos teniendo uno de ellos un dispositivo compañero. Obtenido de [4].

3.2.5. 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:

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*[17].

3.2.6. 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.

3.2.7. 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, $ChainKey = HMAC\text{-}SHA256(ChainKey, 0x02)$.

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

3.2.8. 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. $Chain\ Key, Root\ Key = HKDF(Root\ Key, ephemeral\_secret)$.

3.3. iMessage?

Bibliografía

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

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

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

[4] 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/.

[5] MD5 - Wikipedia. Acceso: 24-05-2023. URL: https://en.wikipedia.org/wiki/MD5.

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

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

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

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

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

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

[12] WhatsApp Encryption Overview. 2023.

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

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

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

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

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

[18] 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/.

[19] F. J. Lobillo. *Teoría de Números y Criptografía.* Granada, 2021.

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

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

[22] 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.

[23] 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.

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

[25] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. *Lecture Notes in Computer Science*, 3494:19–35, 2005. doi:10.1007/11426639_2.