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 de las aplicaciones de mensajería.

2.1. Introducción a la criptografía

Mayormente la información de este apartado ha sido obtenida de [4]

2.1.1. Cifrado y secreto

- $\mathcal{M}$ el conjunto de los mensajes, textos en claro o *plaintexts*,
- $\mathcal{C}$ el conjunto de los criptogramas o *cyphertexts*,
- $\mathcal{K} \subseteq \mathcal{K}_p \times \mathcal{K}_s$ el espacio de clave o *key space*

Un criptosistema 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 $m \in \mathcal{M}$,

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

Fijadas claves $k_p \in \mathcal{K}_p$ y sus correspondiente $k_s \in \mathcal{K}_s$ se definen las funciones de cifrado y descifrado como:

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

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

En la criptografía clásica, también llamada simétrica, se tiene que $\mathcal{K}_p = \mathcal{K}_s$ y $k_s = k_p = k \in \mathcal{K}$, o al menos hay métodos eficientes para conocer k_s a partir de k_p y viceversa. En la criptografía asimétrica, no se conocen métodos eficientes para conocer k_s a partir de k_p.

2.1.2. Objetivos de la criptografía

- **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.

2.1.3. Ataques

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.
- **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.

2.2. El algoritmo Rijndael AES

El algoritmo Rijndael llamado así en honor a sus dos autores Joan Daemen y Vicent Rijmen, es un algoritmo de cifrado por bloques que fue adoptado en octubre de 2000 por el NIST(*National Institute for Standards and Technology*) para su empleo en aplicaciones criptográficas no militares en sustitución del algoritmo *DES* después de un proceso de más tres años en los que se buscaba un algoritmo que fuera potente, eficiente y fácil de implementar.
Está diseñado para manejar longitudes de clave y de bloque variables entre los 128 y los 256 bits y aunque estos sean variables, en el estándar adoptado por el Gobierno de Estados Unidos en 2001 [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 [5].

2.2.1. 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 son:

- **Electronic CodeBook**
  - ***Cifrado ECB***

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

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

- **Cipher-Block Chaining**
  - ***Cifrado CBC***

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

    Dividimos c en $c_{[0]}...c_{[l]}$ con $c_{[i]} \in \mathbb{B}^N$
    Para i=1,...,l hacer
    $m_{[i]} = D_k(c_{[i]}) \oplus c_{[i]}$
    Devolvemos $m_{[1]}...m_{[l]}$
- **Cipher FeedBack**
  - ***Cifrado CFB***

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

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

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

- ***Descifrado OFB***

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

2.2.2. Estructura de AES

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.

2.2.3. Elementos 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 ha 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.

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

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

Siendo B el bloque que queremos cifrar y S la matriz de estado, el algoritmo AES con n 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)
$N_b = 4$(128 bits)	10	12	14
$N_b = 6$(128 bits)	12	12	14
$N_b = 8$(128 bits)	14	14	14

Cuadro 2.3: Número de rodas en función del tamaño de la clave y bloque

2.2.4. Las Rondas de AES

Dado que el algoritmo AES puede aplicarse para longitudes diferentes de bloque y clave, el número de rondas es variables, 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 2.4.
La función inversa será el desplazamiento de las filas de la matriz el mismo número de posiciones pero en el sentido contrario.

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

Cuadro 2.4: Valores de c_i según el tamaño de bloque N_b

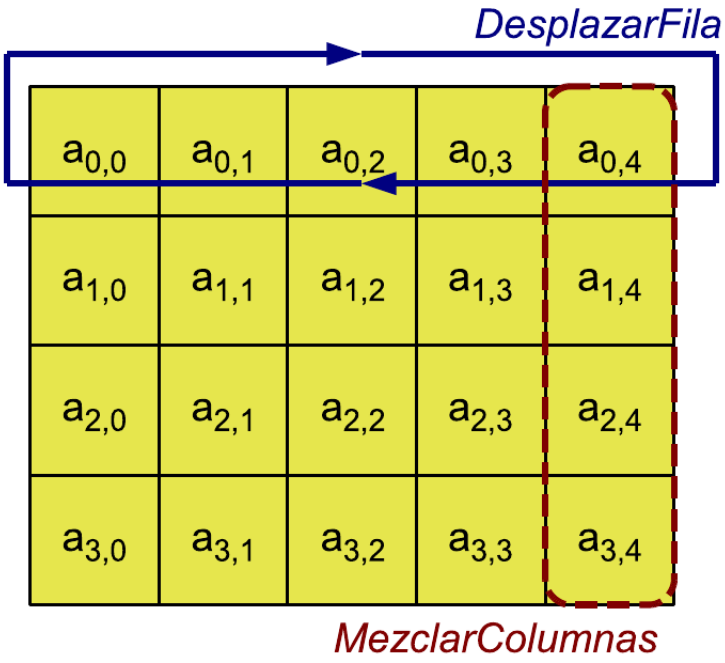

Figura 2.1: Esquema de las funciones *MezclarColumnas* y *DesplazarFila*

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: $d(x) = 0Bx^3 + 0Dx^2 + 09x + 0E$

2.2.5. 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 $GF(2^8)$ correspondiente al valor x^{i-1} módulo $x^8 + x^4 + x^3 + x + 1$

Capítulo 3

Aplicaciones de Mensajería

Me voy a centrar en Telegram, Whatsapp y Facebook Chat, Signal y la de apple.

3.1. Telegram (MTProto)

Referencias: [6] [2]

3.1.1. Descripción general

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.

3.1.2. Resumen de los componentes

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*.

3.1.3. 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 [1].

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	msg_key	encrypted_data
int64	int128	bytes

Encrypted Message: *encrypted_data*

Contains the cypher text for the following data:

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

Unencrypted Message

auth_key_id = 0	message_id	message_data_length	message_data
int64	int64	int32	bytes

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

3.1.4. 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 *SHA1* 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 *SHA1(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: *SHA1(answer)+answer+(0-15 bytes aleatorios)* de manera que la longitud sea divisible por 16, es del tipo *string*.
- *answer_aes_key*: *SHA1(new_nonce + server_nonce) + substr(SHA1(server_nonce + new_nonce), 0, 12)* y es del tipo *string*.
- *tmp_aes_iv*: *substr(SHA1(server_nonce + new_nonce), 12, 8) + SHA1(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:
     - $g\_b = g^b \bmod dh\_prime$
     - *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: *SHA1(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)*.

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 *SHA1(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 SHA1 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 *SHA1(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.

3.1.5. 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 *SHA256(substr(auth_88+x, 32)+plaintext+random_padding)*.
2. Calculamos *msg_key* como *substr(msg_key_large, 8, 16)*.
3. Calculamos *sha256_a* como *SHA256(msg_key+substr(auth_key, x, 36))*.
4. Calculamos *sha256_b* como *SHA256(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

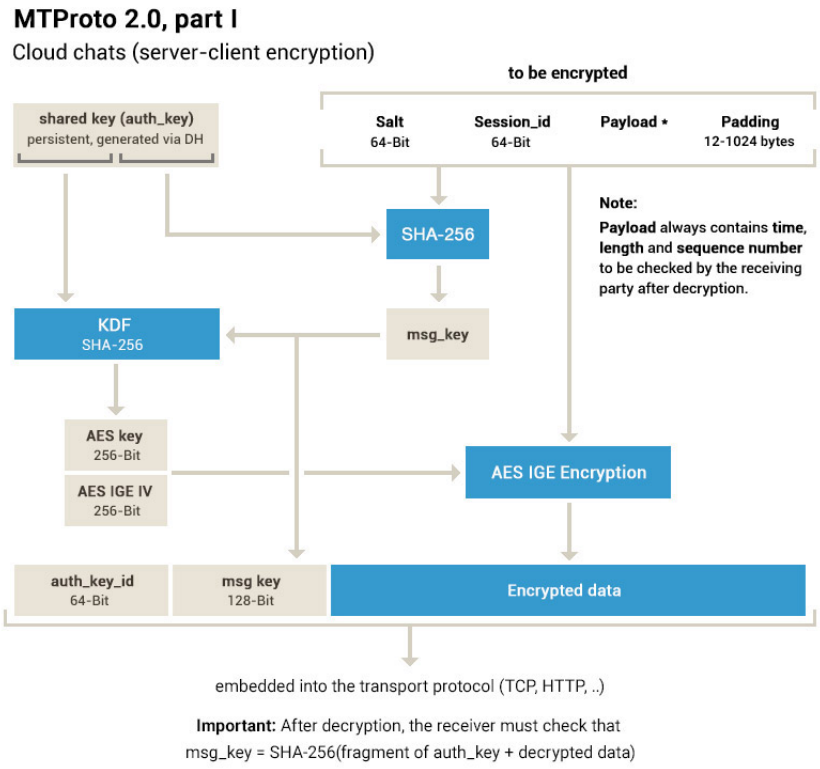

Figura 3.1: Esquema del cifrado de mensajes usado en MTProto 2.0

3.1.6. 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*

Bibliografía

[1] Mobile Protocol: Detailed Description.

[2] MTProto Mobile Protocol.

[3] Federal Information Processing Standards Publication 197 Announcing the ADVANCED ENCRYPTION STANDARD (AES). 2001.

[4] José Luis Gómez Pardo. Criptografía y curvas elípticas. *La Gaceta de la RSME*, 5(3):737–777, 2002.

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

[6] 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*, pages 185–197, 2021.