Universidad Autónoma de Madrid
Escuela politécnica superior

Grado en Ingeniería Informática

Trabajo Fin de Grado

ESTUDIO Y ANÁLISIS DE FIRMAS GRUPALES PARA EL DISEÑO DE PROTOCOLOS DE COMUNICACIONES PRIVADOS, ANÓNIMOS Y JUSTOS.

Autor: Mario Valdemaro García Roque
Tutor: Francisco de Borja Rodríguez Ortiz

Enero 2017

ESTUDIO Y ANÁLISIS DE FIRMAS GRUPALES PARA EL DISEÑO DE PROTOCOLOS DE COMUNICACIONES PRIVADOS, ANÓNIMOS Y JUSTOS.

Autor: Mario Valdemaro García Roque
Tutor: Francisco de Borja Rodríguez Ortiz

Grupo de Neurocomputación Biologica (GNB)
Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Enero 2017

Resumen

Resumen

En la actualidad, la utilización de internet ha crecido debido al mayor uso que las personas dan a los servicios que ofrece. Por ello el tráfico que circula por la red es cada vez más importante, siendo indispensable que las comunicaciones no se vean ralentizadas ni que la información pueda ser obtenida indebidamente por terceros. Por tanto la seguridad es un factor fundamental en la comunicaciones por la red.

Los actuales sistemas de seguridad procuran que toda la información que se intercambia entre dos partes sólo pueda ser conocida por los implicados y no por ninguna entidad externa. Debido a ello han surgido multitud de protocolos y sistemas que procuran que dos partes se puedan comunicar de forma segura. Una elemento esencial de estos sistemas de seguridad es la firma digital, que permite identificar a una de las partes de la conexión.

La firma digital es un sistema ampliamente extendido y se hace uso de ella en los grandes servicios de internet como google, amazon o youtube. Los actuales sistemas de firma llevan muchos años entre nosotros y existen evoluciones de ellos que, además de proporcionar la capacidad de identificación, añaden nuevas funcionalidades interesantes.

Una de las evoluciones de la firma digital ha sido la firma grupal. Este tipo de firma consiste en producir mensajes en nombre de un grupo, es decir, los miembros de un grupo tiene la capacidad de producir mensajes firmados en nombre de esté pero sin que se pueda descubrir quién ha sido el emisor de ese mensaje.

La firma grupal es poco conocida pero tiene un gran potencial práctico, ya sea para anonimizar conexiones de red, controlar el acceso de miembros de un grupo a recursos, tomar acuerdos de forma colegiada de diferentes organizaciones, voto electrónico o consultas ciudadanas anónimas a grupos de población.

En este trabajo se ha realizado un análisis y estudio de la firma grupal y otros sistemas afines, así como una demostración práctica de su utilización mediante el diseño de un protocolo de comunicación segura que hace uso de este tipo de firma como sistema de identificación para miembros de un grupo. El protocolo funciona correctamente y no sobrecarga la red, pero es costoso computacionalmente.

Palabras Clave

firma digital, firma grupal, firma colegiada, protocolos de comunicación, SSL, TLS, firma grupal revocable, firma grupal corta, firma trazable, firma de anillo, firma ciega, pruebas de conocimiento cero, seguridad, privacidad, anonimicidad, anonimato, anonimato justo, revocabilidad, chat de comunicación

Abstract

Nowadays, the internet traffic has enormously increased due to the use that people gives to net services. Its because of this that the importance of the traffic is increasing. One of the most important subjects of net traffic is that only the sender and the recviver knows the content, so the security has also gain importance since the beginning of the internet.

Security systems try to exchange information between two communicators in a secure way, so it cant be eavesdropped by third parties and can only be known by the sender and the receiver. Due to this, the number of security protocols and systems has raised to offer better security. One of the most important is the digital signature, that allows uniquely identify the sender of a message.

Digital signature is a widely used system,that most of the biggest internet services like amazon, youtube or facebook relay on it to provide a secure communication. The currently digital signature protocols are well-known and have evolved in more complex systems adding useful capabilities.

One of these variations is the group signature. This type of signature allows the members of a group to produce measseges on behalf of the group and the capacity that this message cant be traced back to the original sender.

This kind of capability is underused but has great potential due to desirable application in many systems like anonymous net communication, access control of the members of a group to a certain resource, collegiate decisions, electronic voting or for anonymous citizen consultations.

This document analyses the available group signatures schemes and related systems, and gives a practical demonstration of it. For this purpose a secure communication protocol has been designed in order to identify the members of an anonymous group. This protcol works properly and it does not overload the network, but it is computationally intensive.

Key words

digital signature, group signature, collegiate decision, communication protocols, SSL, TLS, traceable group signature, short group signature, traceable signature, ring signature, blind signature, zero knowledge proofs, security, privacy, anonymity, unlink ability, fair anonymity, traceability, communication chat

Agradecimientos

En primer lugar agradecer a mi tutor y a mi padre el haberme guiado por el camino de la criptografía y la seguridad informática, es por ellos que he realizado este trabajo con el que he disfrutado bastante. Gracias también a Jesus Diaz por desarrollar la librería de la que hace uso este trabajo y por solucionarme ciertas dudas concretas sobre las firmas grupales.

Por otro lado gracias a todos mis amigos y compañeros con los que tantas comidas y buenos ratos he compartido, ya que sin ellos todo el tiempo que he empleado en hacer este trabajo hubiese sido mucho más aburrido. Sobre todo, gracias Manu, Angel, Roy y Carlos.

Por último gracias a todos los profesores que he tenido a lo largo del grado y espero que el trabajo que han puesto en mi quede reflejado más allá de este trabajo.

Índice general

Índice de Figuras

Índice de Tablas

Glosario

- **ZKP**: Zero Knowledge Proof, prueba de conocimiento cero.
- **PPT**: Probabilistic polinomial time, los algoritmos propabilisticos de tiempo polinómico son un tipo de algoritmos no deterministas que hacen una elección probabilística sobre cómo resolver un problema.
- **RSA**: Rivest Shamir Adelman, es uno de los primeros criptosistemas de clave pública. En este tipo de criptosistemas existen dos claves distintas que pueden ser usadas tanto para cifrar un mensaje como para descifrar un mensaje que haya sido previamente cifrado con la otra clave.
- **SSL**:Secure Socket Layer.
- **TLS**: Transport Layer Security.
- **https**: http sobre TLS.
- **PKI**: Public Key Infraestructure.
- **DDH**: Decisional Diffie Hellman.
- **LDH**: Linear Diffie Hellman.
- ***q*-SDH**: q-Strong Diffie Hellman.
- **XOR**: operación OR exclusiva.
- **trapdoor**: función trampilla.
- **función de un solo sentido**: tipo de función que difícil de invertir.
- **DHKE**: Diffie-Hellman key exchange. Intercambio de calves mediante Diffie-Hellman.
- **Funcion hash**: tipo de función que dado un valor de entrada de longitud variable produce un código de longitud fija.
- **SHA256**: Es una función de hash que produce un código de 256 bits.
- **handshake**: comúnmente usado en inglés como forma para referirse al proceso de establecimiento de una conexión entre un cliente y un servidor.
- **exp.**: exponenciación.
- **IRC**: Internet Relay Chat
- **Problema P**: problema resoluble en tiempo polinómico mediante un método determinista.
- **Problema NP**: problema resoluble en tiempo polinómico mediante un método no determinista. O que se puede resolver de forma determinista en un tiempo exponencial.

- **Problema NP completo**: problema para el que existe una transformación de cualquier problema NP a él.
- **Método de resolución determinista**: método de resolución que cada vez que se resuelve el mismo problema lo resuelve con mismo resultado y ejecución.
- **Método de resolución no determinista**: método de resolución que cada vez que se resuelve el mismo problema puede variar el resultado y la ejecución.
- **MCD**: máximo común divisor.

1 Introducción

Este trabajo estudia y analiza algunos de los sistemas de firma grupal y sus posibles aplicaciones, enunciando para ello qué es la firma digital y la evolución que ha sufrido desde los esquemas básicos de clave pública, hasta las soluciones más avanzadas cómo firmas grupales. Para demostrar el uso de la firma grupal se ha desarrollado un protocolo de comunicación que hace uso de ella como método de identificación anónimo y justo.

1.1. Motivación

La firma digital es un sistema de identificación que permite comprobar si un mensaje proviene de una cierta entidad/servicio o no. Este tipo de firma está ampliamente extendido en la actualidad, donde un gran número de importantes servicios de Internet hacen uso de ella, ya que permite que no se suplante la identidad de estos servicios.

Los primero trabajos relacionados con firmas digitales están muy ligados a los sistemas de clave pública/privada desarrollados en las décadas de 1970 y 1980 por Diffie y Hellman (Diffie and Hellman, 2006), también por Rivest, Shamir y Adleman (Rivest et al., 1978) y por Merkle (Merkle and Hellman, 1978). Actualmente los sistemas de firma digital se utilizan junto a sistemas de cifrado para conseguir una comunicación privada. Un sistema de este tipo permite a dos o varias partes establecer un canal de comunicación en el que nadie no implicado sea capaz de conocer el contenido del mensaje. Sin embargo, todos estos sistemas se olvidan de un aspecto importante de la privacidad, el anonimato.

El anonimato es una cualidad que impide identificar al emisor de un mensaje. Es decir, ni un observador externo ni el receptor del mensaje podrían conocer la identidad del emisor del mensaje.

Más allá de cómo se comunican los datos, es destacable que actualmente gran parte de las empresas de comercio electrónico, redes sociales, etc., recogen una cantidad inmensa de datos sobre los gustos o patrones de comportamiento de los usuario de servicios con los que posteriormente sacan beneficio, ya sea vendiendo estos datos a otras empresas o usándolos para publicitar a sus propios productos a los usuarios.

Por tanto es deseable para un usuario que no se use la información que genera sin su consentimiento. Es más, está regulado por ley de protección de datos de carácter personal BOE298

(1999) que datos como registros médicos permanezcan anónimos para cualquier tipo de estudios, como estudios clínicos.

Por motivos como estos surgen sistemas que aun haciendo uso de la firma digital, no identifican al emisor del mensaje como una persona sino como miembro de un grupo de personas. Una de las pocas implementaciones de firmas grupales la podemos encontrar en Diaz et al. (2014b), que sera fundamental para este trabajo.

1.2. Objetivos

El objetivo principal que se ha llevado a cabo en este trabajo es un estudio y análisis sobre como generar privacidad y anonimato en los sistemas de información mediante las firmas grupales. Para ello se han propuesto los siguientes hitos:

1. Realizar un estudio sobre las firmas digitales y metodologías afines más comunes para conseguir privacidad, y también estudiar y analizar algunos tipos de firma digital que proporcionan anonimia y anonimato justo.
2. Realizar un estudio y análisis sobre esquemas particulares de firma grupal y sus fundamentos.
3. Implementar una prueba de concepto que haga uso de las firmas grupales como método de identificación en las comunicaciones, en concreto, un protocolo de comunicación para un chat de texto.

1.3. Estructura de la memoria

Para llevar a cabo los objetivos enunciados se ha desarrollado una memoria que se divide en los capítulos siguientes:

- **Estado del arte:** En este capítulo se explica qué son las firmas digitales y como se relacionan con otros sistemas para conseguir comunicaciones privadas. Posteriormente se explican firmas digitales más avanzadas, que es una prueba de conocimiento cero y como se relaciona con la firma digital. Este capítulo recoge parte de los hitos **1** y **2**.
- **Sistema y diseño:** En esta capítulo se detalla el protocolo y el chat que se ha diseñado para poner a prueba el sistema de firma grupal. Este capítulo recoge parte de los hitos **2** y **3**.
- **Desarrollo:** En esta sección se explica qué herramientas se han usado y desarrollado para la creación del protocolo y del chat. Este recoge parte de los hitos **2** y **3**.
- **Pruebas y resultados:** En este capítulo se llevan a cabo una serie de pruebas sobre el protocolo desarrollado que medirán tanto el rendimiento y cómo su correcto funcionamiento. Este capítulo recoge parte del hito **3**.
- **Conclusiones y trabajo futuro:** En este capítulo se exponen los resultados conseguidos y el posible trabajo que se pueda realizar en un futuro.
- **Anexos:** Detalles necesarios para entender el funcionamiento interno de algunas partes del protocolo. Este capítulo ayuda a llevar a cabo los hitos **1**, **2** y **3**.

2

Estado del arte

En este capítulo se enuncian las bases teóricas necearías para entender el trabajo realizado. La sección 2.1 describe que es la privacidad, el anonimato y el anonimato justo, en la sección 2.2 se explica el funcionamiento de la firma digital, en la sección 2.3 se explican algunos tipos de firma mas avanzada, en la sección 2.4 se explica que es una prueba de conocimiento cero y por último, en la sección 2.5 se explican las propiedades de varios sistemas de firma grupal concretos.

2.1. Privacidad, anonimia y anonimato justo

Como se ha comentado en la motivación de este proyecto (ver sección 1.1) los usuarios de los mayores servicios de internet, como Facebook o Amazon, son sensores para estas compañías, pudiendo obtener de ellos una ingente cantidad de información que puede ser usada para para lograr beneficios gracias a la venta de esa información a terceros o el uso de publicidad personalizada. Por tanto, puede ser deseable para los usuarios que dicha información que personal permanezca anónima y privada.

Por un lado la privacidad es una característica deseable para los usuarios, ya que permite que sólo la persona o servicio con el que se están comunicando conozca el contenido de lo que se comunica. Y por otro lado, el anonimato permite al usuario que no se ligue información a su identidad.

A continuación se define más formalmente en qué consisten estas cualidades y que se entiende por anonimato justo.

2.1.1. Privacidad

La privacidad consiste en que lo comunicado entre dos partes, A y B, solo pueda ser conocido por ellas y por nadie más. A lo largo de la historia han existido infinidad de sistemas para intentar garantizar la privacidad de las comunicaciones, desde el cifrado César hasta los actuales sistemas como AES (NIST, 2001), RSA (Rivest et al., 1978).

La privacidad está ligada al cifrado de la información. En la mayoría de las comunicaciones de internet se usa un sistema/protocolo de clave simétrica para cifrarlas, comó pueda ser AES (ver

anexo D). En los sistemas de cifrado simétrico existen dos individuos A y B, que conocen una clave común K, y puede existir una tercera individuo Z, que puede intervenir en la comunicación, pero que no conoce K. El objetivo es que A y B intercambien información sin que Z sea capaz de conocerla; para ello A y B usan la clave K y un sistema de cifrado simétrico que haga uso de dicha clave.

El problema que existe ahora es garantizar que A y B se intercambien la clave K de forma secreta. Para conseguirlo se han desarrollado los sistemas/protocolos de calve pública como Diffie-Hellman (ver anexo C) o RSA (ver anexo B). Estos sistemas están compuestos por una serie de claves privadas, PR_i, y una serie de claves públicas PU_i. La claves están ligadas a cada individuo i, cada clave privada es conocida por un único miembro del sistema, su propietario, y las claves públicas por todos los individuos. Este sistema permite comunicar información secreta a un miembro A cifrando con su clave pública y descifrando con su clave privada. Otra capacidad de este sistema es la de firmar mensajes. Para ello un miembro A cifra un mensaje con su clave privada y cualquier otro miembro del sistema la puede descifrar con la clave pública del miembro A.

Por otra parte es fundamental que B pueda comprobar que la clave pública que tiene es de A y no de otra persona Z la hubiera cambiado intencionadamente. Para resolverlo aparecen las entidades certificadoras, que explicaremos detalladamente, junto al funcionamiento de la firma digital, en la sección 2.2.

2.1.2. Anonimato

Como ya hemos explicado en la motivación (ver sección 1.1), la mayoría de los sistemas/protocolos que aportan privacidad dejan de lado el anonimato, siendo ésta una cualidad importante. Por tanto se definirá más en detalle en qué consiste el anonimato.

El anonimato es una característica que permite a una persona A enviar un mensaje a otra persona B, sin que B se capaz de identificar que el mensaje ha sido remitido por A. Como se puede ver en (Pfitzmann and Hansen, 2009) el anonimato es una propiedad que sólo existe si se asume que A forma parte de un grupo extenso de personas que puede generar contenido anónimo. Se puede decir entonces que el anonimato es probabilisticamente cuantificable, es decir, si hay N personas que pueden producir un mensaje la probabilidad de identificar al emisor original es de $1/N$.

El anonimato se puede conseguir de múltiples formas y en diferentes lugares. Por ejemplo, uno de los anonimizadores más famosos es la red tor (tor, 2016), que intenta crear un sistema de comunicación por internet anónimo desde el emisor hasta el receptor. Otro tipo de anonimización sería la que se usa en las bases de datos de estudios clínicos, que requieren preservar la identidad de los pacientes anónima.

Debido a ello al combinar las cualidades de anonimización y de privacidad en sistemas de comunicación se consigue que observador externo sea incapaz de saber absolutamente nada de la comunicación ni de los comunicadores.

2.1.3. Anonimato justo

En el anonimato justo también produce información anónima, pero si se incumplen una serie de reglas preestablecidas, como insultar en un foro, ese anonimato queda "roto", revelando el origen de esa información. Esta capacidad existe por el mal uso que se pude llegar a hacer del anonimato.

Las primeras menciones al anonimato justo las podemos encontrar en (Chaum and van Heyst, 1991), donde se proponía la firma grupal (ver sección 2.3.1) como medio para implementar el anonimato justo. La firma grupal consiste en que una serie de miembros tienen unas claves, una o varias claves por miembro, que al ser usadas producen un código que identifica al grupo pero no al miembro concreto que ha producido el mensaje.

El anonimato justo implica la existencia de dos propiedades importantes:

- **Revocabilidad:** Es la capacidad de que se pueda revocar una clave, es decir, que una clave no pueda volver a ser usada para firmar mensajes en nombre del grupo.
- **Trazabilidad:** Es la capacidad de poder revelar qué clave ha producido un cierto mensaje. Esta capacidad esta muy ligada a la revocabilidad, ya que para poder revocar una clave es necesario conocerla.

2.2. Firma digital

La firma digital,(Stallings, 2014), es un sistema criptográfico que permite que el receptor B de un mensaje m determine que ha sido producido por un emisor A. Las firmas digitales tienen las siguientes propiedades:

- **Autenticación del origen**: permite identificar al emisor de un mensaje.
- **No repudio**: es la imposibilidad de que el emisor de un mensaje demuestre que ese mensaje no ha sido producido por él.

La mayoría de los sistemas actuales de firma, o los más famosos, se basan en protocolos de clave pública. Un sistema típico sería el de firmas mediante RSA (Stallings, 2014).

2.2.1. Firmas con RSA

Las firma RSA es un tipo de firma digital que hace uso del algoritmo RSA (ver anexo B) y una función de hash, típicamente SHA.

Los actores de este sistema serían una persona A que quiere firmar un mensaje y una persona B que quiere verificar que ese mensaje proviene de A. Para ello A ha generado una clave privada PR_a que sólo conoce él, y una clave pública PU_a que es coincida por B y que puede ser conocida por cualquier persona.

En la figura 2.1 se puede ver el funcionamiento del algoritmo que es como sigue:

1. Dado un mensaje M, A produce su correspondiente hash $H = h(M)$.
2. A encripta H con su la clave privada PR_a produciendo una firma S.
3. Se envía tanto S como M.
4. B desencripta la firma S con la clave pública de A, PU_a.
5. B comprueba que el desencriptado de la firma y el hash del mensaje son el mismo.

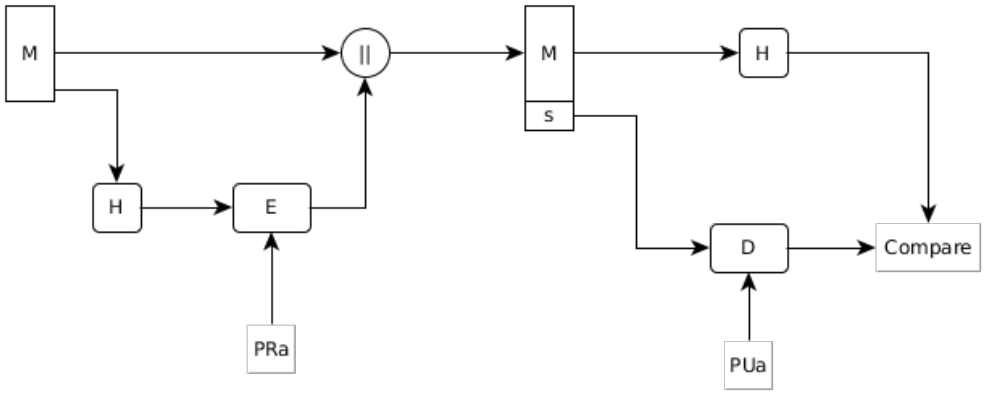


Figura 2.1: Protocolo estándar de firma digital con RSA.

2.2.2. Autoridad certificadora y PKI

Uno de los elementos fundamentales de los sistemas de comunicación segura, como https (rfc2818, 2016) o sistemas similares, es la autoridad certificadora. Una autoridad certificadora es una entidad que se encarga de relacionar una serie de claves públicas con las identidades de sus propietarios. Básicamente, lo que hace es certificar que una clave corresponde a una cierta persona o servicio.

Las autoridad certificadoras forman parte de los sistemas **PKI** (Public Key Infrastructure) (rfc5280, 2016), que son una serie de protocoles y roles que permiten crear una conexión segura y confiable entre dos puntos. Es decir, permite que una comunicación sólo sea conocida por las partes implicadas e identifica al menos a una de las dos partes.

Una autoridad certificadora cumple el rol de ser una entidad confiable, que mantiene un listado de claves públicas relacionadas con el servicio propietario de esa clave. Por ejemplo, si una persona quisiera conectarse a Amazon de forma segura y fiable, Amazon necesitaría tener su propia clave pública certificada por una autoridad de este tipo.

2.2.3. CRL y OCSP

Todo certificado tiene un periodo de validez desde su emisión que suele durar entre 2 y 4 años, una vez sobrepasado el certificado queda revocado. Aparte de por el tiempo un certificado puede ser revocado a petición, por ejemplo si su propietario cree que la clave privada de ese certificado puede haber sido modificada o accedida por alguien que no debía.

Una vez revocado este entra en una lista de revocación de certificados (CRL por sus siglas en inglés). Esta lista esta firmada por una autoridad certificadora y permite a un tercero comprobar si el certificado de un cierto servicio web esta revocado o no. Un protocolo que permite hacer consultas sobre si un certificado esta revocado o no es OCSP, este protocolo evita que la CRL tenga que ser descargada, ya que realiza consultas sobre la CRL de la autoridad certificadora en un servicio que esta provee.

2.3. Firmas digitales avanzadas

Además de las propiedades que ya se ha mencionado que aporta la firma digital, existen sistemas más avanzados de firma que añaden capacidades adicionales. Algunos de estos sistemas

se basan en el uso de protocolos de clave asimétrica como RSA, mientras que otros usan sistemas más complicados, como las pruebas de conocimiento cero (ver sección 2.4).

Algunas de estas firmas avanzadas son la firma grupal y la firma ciega. La firma grupal es un tipo de firma en la que una serie de miembros puede emitir mensajes firmados, no en nombre de ellos, sino en nombre de un grupo al que pertenecen, pero llegado el caso se puede revelar el miembro que produjo la firma. Cabe mencionar que existe un tipo de firma grupal en el que no se puede revelar el miembro que produjo esta firma, que es la firma de anillo (ver anexo H). Por otro lado, las firmas ciegas (ver anexo G) son un tipo de firma en el que el contenido de la firma se ofusca antes de ser firmado.

2.3.1. Firmas grupales

Las firmas grupales son un tipo de sistema de firma digital propuesto en (Chaum and van Heyst, 1991), en el que hay un grupo formado por un serie de miembros con las siguientes propiedades:

1. Sólo los miembros del grupo pueden firmar mensajes.
2. El receptor de la firma puede comprobar que proviene de ese grupo pero no cuál de los miembros la ha producido.
3. Llegado el caso, la firma se puede "abrir" para revelar la identidad del miembro que la produjo.

Además en cierto tipo de sistemas de firma grupal será posible revocar a usuarios para que no puedan volver a firmar en nombre del grupo.

En general este tipo de firmas necesita un grupo de miembros, que son los que pueden emitir firmas, y un administrador del grupo, que dependiendo del protocolo del que se haga uso tendrá unas capacidades distintas, pero en general creará el grupo y podrá revelar la identidad de un firmante.

La posibilidad de "abrir" las firmas es importante en entornos en los que es deseable que los miembros tengan anonimato pero no puedan utilizarlo para actuar en contra de los intereses del grupo. Por ello es importante definir en qué casos se podrá revelar la identidad de un firmante, es decir, tiene que haber un cierto tipo de contrato con los miembros del grupo que fije su responsabilidad si incumplen alguna de las normas.

Un caso ilustrativo se propuso en (Chaum and van Heyst, 1991), en él una compañía da acceso a sus empleados a una serie de impresoras, pero no liga los documentos que se están imprimiendo con los empleados. Sin embargo la empresa quiere que si alguno de los empleados hace un uso excesivo de las impresoras se pueda saber el empleado incumplido para imputarle los costes.

La primera vez que se presentó un sistema de este tipo se hizo con varios protocolos en los que había una autoridad de confianza Z, que en la mayoría de los casos era la que creaba los grupos.

Firmas basadas en múltiples claves RSA

Este sistema se propuso en (Chaum and van Heyst, 1991) como un modelo sencillo de firmas grupales. Está compuesto por una entidad Z que es la creadora del grupo y una serie de miembros P_i que son capaces de emitir una firma en nombre del grupo. La firma la comprobará un verificador V para saber si un cierto mensaje proviene del grupo.

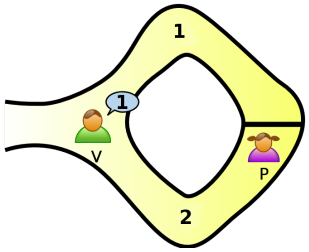


Figura 2.2: Cueva.

En este sistema Z crea una serie de claves públicas con sus correspondientes claves privadas, posteriormente Z publica las claves públicas en un directorio de libre acceso y reparte a los miembros P_i una serie de claves privadas.

Los miembros firman los mensajes con la clave privada y la desechan después de haber firmado, ya que se podría relacionar varios mensajes firmados con la misma clave pública. En caso de que una firma genere una disputa, al conocer Z todas las claves privadas, se pude revelar que miembro emitió la firma.

Este sistema tiene algunos inconvenientes frente a otros sistemas de firma grupal, como que el número de mensajes que emita un miembro del grupo está limitado al número de claves privadas sin usar que tenga.

Firmas basadas en pruebas de conocimiento cero

Existen multitud de sistemas que hacen uso de las pruebas de conocimiento cero (Sección 2.4) para comprobar si una firma es válida o no. La idea principal de aplicar este tipo de pruebas a los sistemas de firma grupal es que no revelan información sobre la identidad del emisor del mensaje.

Entre los múltiples protocolos de firma grupal algunos hacen uso de una autoridad confiable Z que genera las claves de los usuarios, y otros no.

2.4. Pruebas de conocimiento cero

Las pruebas de conocimiento cero (ZKP por sus siglas en inglés), (Prof. Dr. Mark Manulis, 2012), son un tipo de pruebas que se realizan entre dos partes: P que hace una declaración y V que verifica que esta declaración sea cierta, es decir, un **probador** y un **verificador**. La característica más importante de estas pruebas y que les da ese nombre, es que no se revela información como tal, más allá de confirmar que el probador P tiene el conocimiento que dice tener. Por tanto al no revelar información concreta es un buen mecanismo de identificación anónima.

2.4.1. Un ejemplo sencillo

Uno de los ejemplos más sencillos para explicar cómo funcionan este tipo de pruebas fue propuesto por (Quisquater et al., 1989). En resumen, tenemos estas condiciones previas, que se pueden ver en la figura 2.2:

1. Existe una entrada a una cueva circular en la que hay una puerta en el camino que sólo se abre si se conoce una clave.

2. P conoce esta clave pero no quiere revelarla.
3. V quiere saber si P conoce esta clave.

La prueba de conocimiento cero sería demostrar a V que P sabe la clave, pero sin revelarsela.

Un posible caso sería el siguiente:

1. P entra en primer lugar a la cueva por el camino 2 sin que V sepa que ha entrado por él. V nunca sabe porque camino entra P. P debe escoger el camino por el que entra de forma aleatorioa.
2. V entra después en la cueva y le pide a P que salga por el camino 1. V escoge el camino también de forma aleatoria.
3. P sale por el camino 1 gracias a que conoce la clave.

Existe el caso de que P no sepa la clave y sin embargo salga por el camino por el que V le pide que salga, debido a que ha entrado por ese camino, esto es un problema. Para solucionarlo P y V repiten esta prueba numerosas veces, si en alguna de estas ocasiones P no es capaz de salir por el camino correcto, entonces P no sabrá la clave.

2.4.2. Funcionamiento para la computación

El ejemplo anterior es muy sencillo, pero deja abiertas dudas sobre la posible implementación en ordenadores.

Lo primero que hay que entender es que tanto **P** como **V** no son personas como tal sino ordenadores, computadores o **máquinas de Turing (MT)**, a lo largo de este trabajo denominaremos como MTs. El motivo para que P y V sean MTs es que necesitamos la capacidad computacional que tienen ya que realizaran calculaos complejos.

El siguiente factor a tener es cuanta es **¿qué demuestra P a V?** Sobre esto, lo primero que hay que explicar es que **P tiene una solución para un problema matemático complejo de computar** y que es inviable obtener esa solución en el tiempo que dura el intercambio entre P y V. Existen multitud de problemas de para realizar estas de pruebas. Los fundamentales para entender los detalles más profundos de este trabajo son las suposiciones que se encuentran en el anexo J. Además, hay otros problemas para los que existen ZKPs interesantes, como puede ser el problema de los tres colores (ver anexo K.1).

Recapitulando, para realizar una prueba de conocimiento cero se necesita:

- Un verificador V, que es una MT.
- Un probador P, que también es una MT.
- Un problema computacionalmente inviable de resolver si no se conoce la solución. Es decir, un problema tan complejo que el mejor método conocido no permita resolverlo en el tiempo que dura una comunicación, a no ser que se conozcan ciertos valores de este problema. Son ejemplos de este tipo de problemas: obtener los factores primos de un número (ver anexo E), computar el logaritmo discreto de un número (ver anexo F) y otros más (ver anexo J).
- Los valores que permiten resolver rápidamente el problema, es decir, la solución.
- Y por último, un método para demostrar que se posee una solución del problema, pero que no la revele. Este método será una prueba de conocimiento cero.

Generalmente ninguna de las pruebas es cierta matemáticamente ya que se trata de pruebas probabilísticas, y por tanto, existe una pequeña posibilidad de que se pueda engañar al verificador. Para evitar esto se procura que el error provocado por la prueba sea ínfimo realizando múltiples veces este tipo de pruebas.

2.4.3. Pruebas interactivas, no interactivas y propiedades

El ejemplo de la cueva (ver sección 2.4.1) sería un caso claro de una prueba **ZKP interactiva**, estas pruebas, a diferencia de las no interactivas, dependen una entrada proporcionada por V para que P de la salida correcta, en el ejemplo sería el caso en el que P ya está dentro de la cueva y V le pide que salga por uno de los dos caminos. Las **ZKP no interactivas** son aquellas en las que no se necesita una interacción entre P y V, ya que la información enviada es suficiente por ella misma como para demostrar que P tiene el conocimiento que afirma tener.

- Completitud: Si la prueba es cierta el verificador se dará por convencido de que así es.
- Validez: Si la prueba es falsa no es posible convencer al verificador de que sea cierta.
- Conocimiento cero: Si la prueba es cierta no es viable obtener por parte del verificador el conocimiento de esa prueba.

2.5. Firmas grupales avanzadas

Existen sistemas de firma grupal que hacen uso de las pruebas de conocimiento cero como método de firma. Hay varias versiones de cómo crear un sistema de este tipo, algunos están basados en el mismo problema pero lo resuelven de forma diferente. También hay desarrollos para estas firmas aportan funcionalidades adicionales, como la reclamación de mensajes.

2.5.1. Firmas grupales revocables

Las revocación de una firma es un proceso que permite al administrador de un grupo averiguar la calve de un miembro, y dado el caso marcar esa clave como inválida. Este trabajo se hace uso de las firmas revocables. Estos sistemas tienen dos propiedades fundamentales:

- **Trazabilidad**: consiste en que dada una firma sobre un mensaje, un administrador de grupo es capaz de descubrir mediante un proceso matemático que clave ha producido esa firma.
- **Revocabilidad**: consiste en que un administrador de grupo pude marcar una cierta clave como inválida para producir firmas.

Los esquemas que se usan para ello son los que aparecen en la librería **libgrubsig** del artículo (Diaz et al., 2014b), donde se propone una implementación de tres esquemas con capacidades similares. En esta sección se explicara el esquema KTY04 (Kiayias et al., 2004), y en la sección 2.5.2 se explican BBS04 (Boneh et al., 2004) y CPY06 (Choi et al., 2006), el nombre de estos esquemas corresponde a las iniciales de sus autores y año de publicación.

KTY04

KTY04 (Kiayias et al., 2004) es un sistema de firmas revocables en el que la revocación la hace una entidad "central", que puede ser la misma que ha generado el grupo u otra distinta, dependiendo de cómo se quiera implementar el sistema.

La ZKP en la que se basa este sistema depende de las suposiciones criptográficas DDH (Anexo J), el logaritmo discreto (Anexo F) y strong-RSA (Anexo E) para llevar a cabo la prueba. Las funcionalidades propuestas en este sistema son:

- `Setup()` ejecutada por el administrador del grupo, produce un valor publico pk_{GM} y un valor privado sk_{GM} con el que se generaran las claves.
- `Join()` ejecuta por el administrador del grupo, dado el valor privado sk_{GM} genera un certificado $cert_i$ para el usuario i y se guardan las operaciones en una base de datos privada `Jtrans` controlada por el administrador.
- `Identify()` ejecutada entre un verificador y un probador, se lleva a cabo una ZKP sobre el certificado del probador $cert_i$. Básicamente, produce una firma.
- `Open()` ejecutada por el administrador, dado el valor sk_{GM}, una prueba de identificación y la base de datos `Jtrans`, revela identidad del probador de la prueba de identificación.
- `Reveal()` ejecutada por el administrador del grupo, dado el resultado de la función `Join` para un usuario i revela la "clave" que esta usando.
- `Trace()` ejecutada por el administrador del grupo, dado el resultado de la función `Identify` y el resultado de la función `Reveal` comprueba si una cierta firma ha sido producida por esa clave.
- `Claim()` ejecutada por un verificador y un probador, está función liga el resultado de la operación `Identify` al probador y comprueba que realmente ha sido el probador el que ha generado la firma .

Revocación

Para revocar una firma lo que hace el sistema es usar los resultados de la operación `join`, donde se genera un archivo con una parte de la clave generada para cada miembro `Jtrans`. Posteriormente se usa la operación `open` que usará el resultado de `identify`, es decir, la firma de un mensaje, y el archivo que hemos mencionado que produce `join` para identificar qué entrada ha generado la firma. Finalmente se usa `trace` con la entrada anterior para producir la clave que ha generado la firma.

Tamaño de la firma y eficiencia

Este sistema genera una firma de unos 1.206 bytes, que es considerablemente grande para una firma, sobre todo teniendo en cuenta que una firma que haga uso de claves RSA de 2048 bytes produce una firma de 256 bytes. Y, por otro lado, el sistema lleva a cabo 19 exponenciaciones para el proceso de firma y 17 para el proceso de verificación. Es un sistema bastante costoso computacionalmente y requiere bastantes recursos, sin embargo el proceso de generación del grupo es poco exigente en comparación con otros sistemas de firma grupal.

2.5.2. Firmas grupales cortas

Como hemos visto antes KTY04 tiene el problema de que genera unas firmas bastante grandes y bastante costosas de producir. Mirándolo desde el punto de vista de este trabajo, es una condición no deseable puede sobrecargar la red de comunicación y porque exige más computo al emisor de las firmas.

Con el objetivo de producir una firma más corta surgen los esquemas de firma BBS04 y CPY06, que producen firmas mucho más cortas y menos costosas computacionalmente. La desventaja que tienen es qué el problema en el que se basan es más complicado que el que usa KTY04. Por tanto, producir una solución para este problema es más costoso, pero al ser un proceso que se realiza únicamente al establecer el grupo, no es un detalle demasiado importante.

CPY06

CPY06 (Choi et al., 2006) es un sistema de firmas grupales basado en una ZKP para la verificación sobre el problema q-SDH (Anexo J). El tipo de ZKP que se realiza es parecida a la explicada en el anexo K.2, pero se hacen otro tipo de operaciones por parte del verificador V y del probador P a la hora de transmitir la información sobre el secreto de P y sobre cómo comprobar que P realmente conoce este secreto.

Este sistema necesita de un administrador del grupo Z, que será quien genere las condiciones iniciales del grupo y una serie de miembros, que tendrán sus claves para firmar mensajes.

Las funcionalidades del sistema son las siguientes:

- `Setup`(1^k) es la operación con la que el administrador genera las condiciones iniciales del grupo, que se reflejarán en la clave del grupo Y, que es pública y la clave del administrador S, que es privada. Además se escoge una función de hash.
- `Join`(Y) `Iss`(Y, S) son las operaciónes para crear un miembro. Con la operación `Join` un usuario se une al sistema y cuando hay una solicitud de unión el administrador da un certificado al solicitante aplicando `Iss`. El nuevo miembro i recibirá un certificado $cert_i$, que junto con un secreto sec_i podrá usar para firmar mensajes. Los valores $cert_i$ y sec_i formaran su clave.
- `Sign`$(m, Y, cert_i, sec_i)$ es la operación para generar una firma σ sobre un mensaje m de unos 362 bytes de longitud aproximadamente. Para firmar el usuario usará su clave $(cert_i, sec_i)$ y la clave de grupo Y.
- `Verify`(m, σ, Y) verificará si una firma σ ha sido generada a partir del mensaje m por un miembro del grupo, se usa la clave grupal Y.
- `Open`(σ, Y, S) esta operación permite al administrador del grupo abrir una firma para conocer quién la emitió. Junto con la función `Reveal`$(i, transcript)$ se podrá identificar al miembro.
- `Trace`(σ, C, Y) es una función que sirve para verificar si una cierta firma σ está siendo usada por un usuario que ha generado una información C.
- `Claim`(σ, Y, sec_i) sirve para que un miembro del grupo i informe de que una firma σ ha sido generada por él.
- `ClaimVerify`(σ, τ, Y) esta función verifica que el resultado de `Claim` τ es correcto.

operación	**BBS04**	**CPY06**
`sign`	8 exp.	6 exp.
`verify`	6 exp.	7 exp.
longitud	192 bytes	362 bytes

Cuadro 2.1: BBS04 CPY06 eficiencia y tamaño.

BBS04

BBS04 (Boneh et al., 2004), es un sistema de firma grupal bastante parecido a CPY06, ya que para realizar la ZKP se basan en resolver el mismo problema q-SDH. Sin embargo las funcionalidad de BBS04 no incluye la capacidad de realizar las pruebas de `Claim` y `ClaimVerify`, pero produce firmas más pequeñas, de unos 200 bytes. En la tabla 2.1 se detallan más las diferencias de ambos sistemas, pero como se puede ver, en tiempos de computó son muy similares.

Ambos protocolos producen firmas comparables a la firma con RSA, ya que su longitud es de 256 bytes para claves de 2048 bits y de 512 bytes para claves de 4096 bits, actualmente usar claves de 2048 bits se considera seguro (NIST, 2016). Por tanto usar firmas grupales esta justificado desde el punto de vista de la congestión de la red.

3

Análisis y diseño del sistema

En este capítulo se ha diseñado un protocolo de comunicación seguro basado en firmas grupales y para probar su correcta funcionamiento se ha decidido construir un chat de texto tipo IRC, además de un sencillo sistema de revocación de usuarios para probar esta propiedad de las firmas grupales. En las sucesivas secciones se detallan las decisiones tomadas respecto al diseño del protocolo de comunicación 3.1, del chat 3.2 y del sistema de revocación 3.3, así como los requisitos funcionales y los no funcionales 3.4 y los casos de uso 3.5.

3.1. Estructura del protocolo para la creación de un canal de comunicación anónimo

El protocolo que se ha diseñado para este trabajo está basado en estándares para conexiones seguras, principalmente inspirado en el funcionamiento de TLS/SSL (rfc5246, 2016). Estos protocolos hacen uso de una autoridad certificadora, protocolos de clave asimétrica y protocolos de clave simétrica, todo ello produce canal de comunicación segura.

En este trabajo se usa un esquema parecido SSL (ver anexo L), pero que añade las firmas grupales como método de identificación y de firma de mensajes, sin embargo no se hace uso de claves certificadas (rfc5280, 2016).

El objetivo de este protocolo es conseguir una comunicación privada entre cliente y servidor verificando la identidad de ambos, usando firmas digitales para identificar al servidor y firmas grupales para identificar al cliente. Para ello el sistema tendrá que contar con los siguiente mecanismos:

- Un sistema de firma digital para el servidor. Se usará el sistema de firma basado en RSA (ver sección 2.2.1), con el que se podrá comprobar que los mensajes proceden del servidor.
- Un sistema de firma grupal para el cliente. Se ha decidido usar el sistemas de firma corta revocable CPY06 (sección 2.5.2), ya que produce unas firmas cortas y tiene un buen rendimiento computacional. El servidor usará estas firmas para verificar que un mensaje ha sido generado por un miembro del grupo.

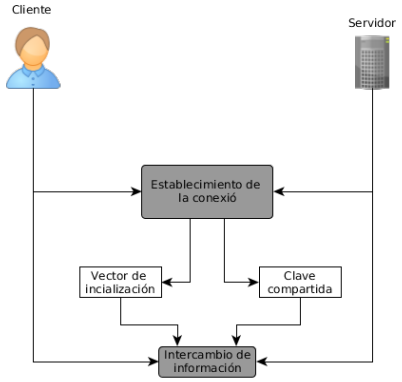


Figura 3.1: Proceso de comunicación entre un cliente y un servidor.

- Un sistema de intercambio de claves. Cliente y servidor se pondrán de acuerdo en los valores iniciales que se usarán en un cifrado simétrico. Para ello se usará el protocolo de intercambio de claves de Diffie-Hellman Key Exchange (DHKE) (anexo C).
- Un sistema de clave simétrica. Cliente y servidor lo usarán para intercambiar información eficientemente. Para ello se utiliza el cifrado AES con claves de 256 bits y en el modo cbc (Anexo D).

El motivo para que el servidor no use firmas grupales es la necesidad de saber que realmente el servidor es el servidor y no un miembro del grupo, ya que sí no, otro miembro podría falsear las comunicaciones haciéndose pasar por el servidor, además de que la firma RSA es más eficiente que cualquiera de los protocolos de firma grupal que se han propuesto. Un resumen del funcionamiento del protocolo se puede ver en la figura 3.1. A continuación se detallan las dos partes principales del protocolo diseñado, el establecimiento de la conexión y la comunicación por un canal seguro.

3.1.1. Establecimiento de conexión

El protocolo de establecimiento de conexión o handshake se realiza entre el servidor y un miembro del grupo. Las condiciones previas para que este sistema funcione y sea seguro son:

Precondiciones

- Que un administrador del grupo Z lo haya creado y haya repartido al menos una clave para un miembro del grupo P_i.
- Que el servidor haya creado sus claves pública y privada mediante RSA y haya distribuido su clave pública.

El protocolo de handshake consistirá por lo tanto en intercambiar la clave y el vector de inicialización que se usarán en AES realizando las operaciones de la figura 3.2 y que se enumeran a continuación:

Protocolo de establecimiento de conexión

1. El usuario envía un mensaje para iniciar la conexión, `hello`.

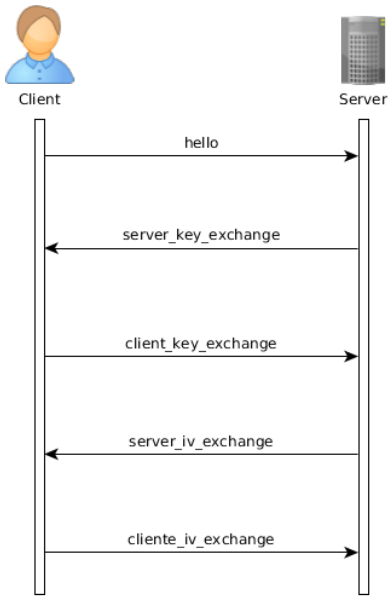


Figura 3.2: Protocolo de establecimiento de conexión entre un cliente y un servidor.

2. El servidor responde con un mensaje firmado con su clave RSA, que contiene la clave pública del protocolo de intercambio DHKE, `server_key_exchange`.

3. El cliente comprueba que la firma se válida y a partir de los valores recibidos genera su clave pública y privada y envía al servidor su clave pública firmada con su clave grupal, `client_key_exchange`. En este punto el cliente ya conoce la clave de AES 256 cbc K.

4. El servidor comprueba que la firma recibida sea válida y que la clave que la ha producido no esté revocada y deriva la clave secreta K de estos valores.

5. Cliente y servidor repiten los dos pasos anteriores para obtener el vector de inicialización IV y dan por exitoso el establecimiento de la conexión.

En este protocolo hay dos partes fundamentales, el intercambio de claves por parte del servidor y el intercambio por parte del cliente. Para ello se va a explicar más en detalle qué operaciones se realizan por parte del cliente y cuáles por parte del servidor. El intercambio del vector de inicialización es análogo al el intercambio de claves.

Las operaciones que se realizan tanto en cliente como en servidor cuando el servidor inicia el intercambio `server_key_exchange`, se pueden ver en la figura 3.3 y se resumen en:

1. `genDHkeys`, el servidor genera las claves pública y privada del protocolo DHKE, `SprivDHkey` y `SpubDHkey`.

2. `signRSAmsg`, el servidor firma la clave `SpubDHkey` produciendo el valor `Ssig`.

3. El servidor concatena firma `SpubDHkey` y clave `Ssig` en un mensaje `Smsg` y lo envía al cliente.

4. El cliente separa la clave `SpubDHkey` y la firam `Ssig` del mensaje que recibe `Smsg`.

5. `checkRSAsign`, el cliente comprueba que la firma es correcta.

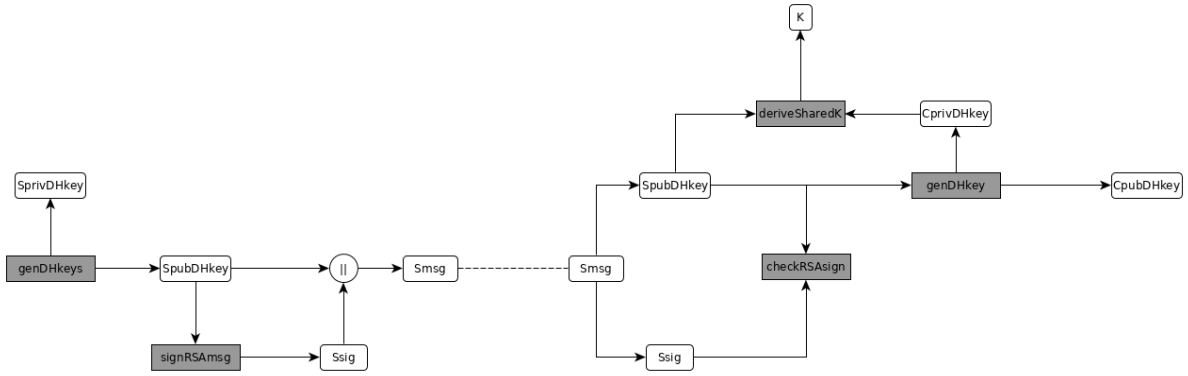


Figura 3.3: Intercambio de la clave desde el servidor hasta el cliente. En esta parte el servidor genera las valores iniciales de la clave que cliente y servidor compartirán.

6. `genDHkey`, el cliente genera sus claves `CprivDHkey` y `CpubDHkey` a partir de la clave pública del servidor `CpubDHkey`.
7. `deriveSharedK`, el cliente deriva el secreto compartido `K` a partir de `CprivDHkey` y de `SpubDHkey`.

Una vez hecha esta parte satisfactoriamente el cliente responde con `client_key_exchange`, que como se puede ver en la figura 3.4 consiste en:

1. Una vez generadas las claves anteriores (`SprivDHkey` , `SpubDHkey` , `CprivDHkey` y `CpubDHkey`), el cliente firma con su clave de miembro `CpubDHkey`, produciendo `Csig`.
2. El cliente concatena `CpubDHkey` y `Csig` produciendo el mensaje `Cmsg`, que envía al servidor.
3. El servidor recibe `Cmsg` y lo separa en `CpubDHkey` y `Csig`.
4. `verifyGSsign`, el servidor verifica que la firma sea correcta a partir de `CpubDHkey` y `Csig`.
5. `traceGSsign`, el servidor verifica que la clave no haya sido revocado a partir de `CpubDHkey` y `Csig`.
6. `deriveSharedK`, el servidor deriva la clave `K` a partir de `SprivDHkey` y de `CpubDHkey`.

3.1.2. Canal de comunicación

Una vez establecidas la contraseña y el vector de inicialización ambas partes son capaces de enviar información de forma privada, y en caso de ser uno de los miembros del grupo, también de forma anónima. En este punto cliente y servidor enviarán información de forma distinta. El funcionamiento del protocolo por parte del servidor se puede ver en la figura 3.5 y consiste en:

1. Dado un mensaje m el servidor lo cifra mediante la clave simétrica K y el vector de inicialización correspondiente a ese momento del intercambio[1] IV_i, produciendo un mensaje cifrado *enc* que se envía al miembro del grupo y por último actualiza el valor de IV_{i+1} con el valor de *enc*.
2. El miembro del grupo correspondiente recibe el mensaje cifrado *enc*, lo desencripta usando la clave simétrica K y el vector de inicialización IV_i y actualiza el valor de IV_{i+1}.

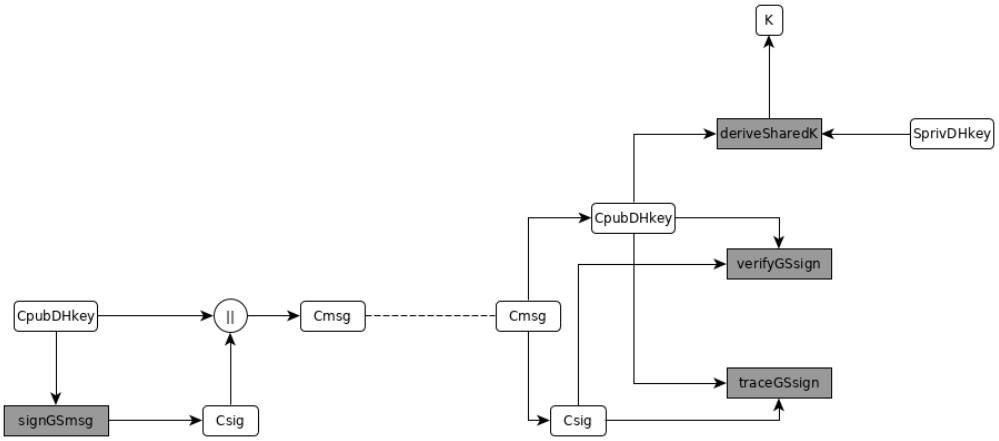


Figura 3.4: Intercambio de la clave desde el cliente hasta el servidor. En esta parte el cliente termina de enviar los valores de la clave compartida para que el servidor pueda generarla.

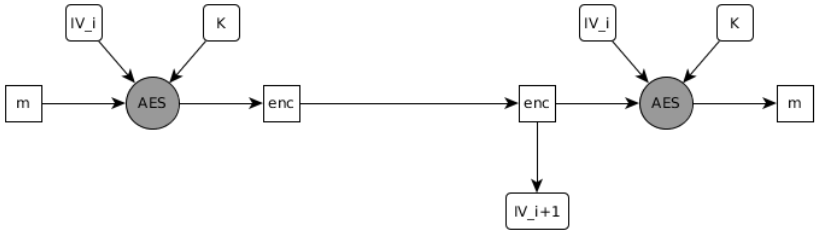


Figura 3.5: Envío de datos por el servidor.

El funcionamiento del protocolo por parte del usuario se puede ver en la figura 3.6 y consiste en:

1. Dado un mensaje m el miembro del grupo k genera una firma grupal, Sig_k la concatena con el mensaje, todo ello lo encripta con la clave K y el valor IV_i produciendo el mensaje cifrado *enc*, envía *enc* al servidor y actualiza el valor de IV_{i+1}.

2. El servidor recibe *enc*, lo desencripta con la clave K y el valor IV_i, obteniendo el mensaje m y la firma grupal Sig_k, comprueba que la firma ha sido generada por un miembro del grupo[2] y que corresponde al mensaje enviado y actualiza el valor IV_{i+1}.

3.2. Estructura del chat

El chat desarrollado es un implementación muy simple de un chat IRC (Internet Relay Chat), ya que lo que se quiere comprobar en este trabajo es el correcto funcionamiento de los protocolos de firmas grupales como sistema de firma anónima, por tanto el chat aquí desarrollado tiene unas funcionalidades muy limitadas. A continuación pasaremos a explicar los elementos y funcionalidades del chat.

[1]Recordar que el esquema cbc se retroalimenta de los mensajes anteriores para el vector de inicialización, por lo que se usarán los mensajes que se cifran como sucesivos vectores de inicialización.

[2]Es importante mencionar que en este punto ya no se comprueba si la clave esta revocada, el principal motivo para ello es la eficiencia. Comprobar si una firma está revocada es una operación bastante costosa y redundante, al haber sido ya comprobada durante el establecimiento de la conexión sí esta revocada.

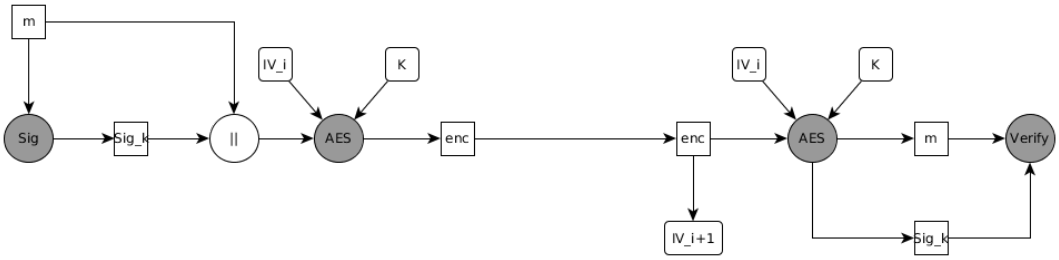


Figura 3.6: Envío de datos por el usuario.

3.2.1. Elementos del chat

Usuarios

Este chat, como la mayoría, se compone de varios usuarios/clientes que pueden comunicar mensajes al resto de los usuarios que se encuentren conectados. Todos los usuarios tendrán un **nick** que les identifica nominalmente, este nick será una cadena de 7 caracteres que se podrá modificar.

Los usuarios podrán enviar mensajes, conectarse al chat, desconectarse del chat y cambiar de nick.

Servidor

El servidor tiene la función de permitir que los usuarios se conecten a él, una vez conectados si un usuario manda un mensaje el servidor se ocupará de transmitir el mensaje a todos los usuarios, incluyendo al origen como confirmación de que el mensaje ha sido recibido.

Otras de las funciones del servidor será ocuparse de mantener la conexión, es decir, verificar que los usuarios siguen realmente conectados y no están ausentes o ha habido algún problema de conexión. Para ello podrá mandar una serie de mensajes a los clientes que para verificar que están conectados.

Canal

A diferencia de los típicos IRCs que tienen varios canales donde existen grupos distintos de usuarios que se comunican entre ellos, en este chat solo existe un canal principal donde cualquier usuario que se haya conectado puede escribir y recibir mensajes.

3.2.2. Comandos

Usuarios y servidor tendrán una serie de comandos que se enviarán entre ellos para llevar a cabo las funcionalidades del chat. Los comandos tendrá distintos efectos en función de si el comando lo recibe un usuario o el servidor. A continuación explicamos los comandos que hay y los distintos efectos que producen en usuario y servidor.

NICK

Un usuario puede enviar este comando en dos situaciones, en caso de que esté realizando el proceso de conexión, donde el usuario enviaré el nick con el que se quiere registrar y para

cambiarse de nick.

Un usuario que recibe el comando nick informará de que un usuario se ha cambiado de nick y por tanto en este comando estará contenida la información del nick anterior y el nuevo.

Un servidor que recibe el comando nick informará a los usuarios de que un nuevo usuario se ha conectado o que un usuario se ha cambiado de nick.

MSG

Un usuario que genere este comando querrá comunicar un mensaje al resto de usuarios. Para ello manda su nick y el mensaje que quiere comunicar.

Si un usuario recibe este comando significa que un cierto usuario ha comunicado un mensaje o le confirma que su mensaje ha sido reenviado.

Un servidor que recibe este comando reenvía el mensaje a todos los usuarios

DSISCONNECT

Un usuario envía este comando cuando se quiere desconectar.

Si un usuario recibe este comando significa que el servidor ha perdido la conexión con el o que un usuario se ha desconectado.

Un servidor que recibe este comando desconecta al usuario que lo haya producido e informa al resto de usuarios de la desconexión.

PING

Si un usuario recibe este comando debe responder con el comando PONG, de ello se ocupará la interfaz del cliente.

Un servidor genera este comando para comprobar la conexión con un usuario. Si recibe este comando no actúa.

PONG

Un servidor recibe este comando como confirmación de la conexión por parte de un usuario.

Un cliente no actúa en caso de recibir este comando.

3.3. Sistema de revocación

El sistema de revocación planteado para este trabajo es bastante simple y sirve para comprobar el correcto funcionamiento de esta capacidad de las firmas grupales. El funcionamiento consiste en ver el contenido de los mensajes que los usuarios se envían entre ellos y si se encuentra algún mensaje que contravenga alguna de las normas del chat, como por ejemplo algún insulto, se revoca la clave del usuario que ha generado ese mensaje y se le desconecta.

3.4. Subsistemas, requisitos funcionales y no funcionales

Según lo que hemos visto anteriormente se puede dividir el sistema en varios subsistemas. Existen tres subsistemas principales, el subsistema de conexión, el subsistema de chat y el subsistema de revocación.

3.4.1. Subsistema de conexión

Este sistema se encarga de que la conexión entre dos puntos sea segura. Los requisitos funcionales de este sistema son:

- **RF1: Orden de conexión.** Los mensajes enviados por un usuario llegaran de forma ordenada. TCP se encarga de ello.
- **RF2: Privacidad en la conexión.** Todo lo que se comunique entre usuarios y servidor solo será conocido por ellos y por nadie más y aunque se intercepten las comunicaciones no se podrá revelar su contenido. Para ello se desarrolla el protocolo propuesto.
- **RF3: Conexión segura con el servidor.** Los usuarios deben poder conectarse con el servidor sin temor a que se esté falseando la identidad del servidor. Que un cliente tenga la clave pública del servidor asegura esto.
- **RF4: Conexión segura de los miembros del grupo.** Sólo los miembros del grupo con una clave válida pueden conectar con el servidor. Las firmas grupales se encargan de ello.
- **RF5: Mensajes anónimos.** Los mensajes enviados que cumplan las reglas del chat no revelarán información sobre qué miembro ha enviado el mensaje. Las firmas grupales se encargan de ello.
- **RF6: Conexión persistente.** La conexión entre los usuarios y el servidor no debe perder mensajes. Para ello se usa un el sistema de envío de mensajes TCP y se realiza un control de conexión por parte del servidor.

3.4.2. Subsistema de chat

El subsistema de chat se encargará de que los usuarios puedan comunicarse entre ellos. Los requisitos funcionales para este sistema son:

- **RF7: Envió de mensajes.** Los usuarios son capaces de comunicarse entre ellos satisfactoriamente. Los programas de cliente y servidor del chat se encargarán de ello.
- **RF8: Elegir un apodo.** Los usuarios son capaces de elegir el apodo que crean conveniente. Clientes y servidor del chat se encargarán de ello.
- **RF9: Desconexión.** Un usuario es capaz de desconectarse en el momento que desee. Los programas de clientes y servidor del chat se encargarán de ello.

3.4.3. Subsistema de revocación

Este subsistema se encarga de que detectar mensajes que contravengan las normas del grupo. El requisito funcional de este sistema es:

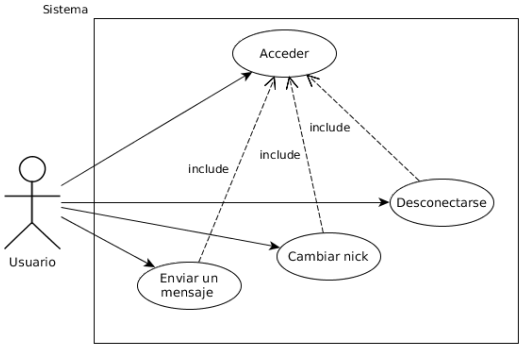


Figura 3.7: Diagrama de casos de uso.

- **RF10: Respeto entre usuarios.** En caso de incumplir las normas de conducta del chat se podrá expulsar al usuario que incumplió las normas. El sistema de revocación propuesto, que se encuentra en el servidor, asegura esto.

3.4.4. Requisitos no funcionales

Por otro lado el sistema tiene que cumplir con los siguientes requisitos no funcionales:

- **RNF1: Tiempo de envío de mensajes.** El tiempo de envío de mensajes no debe ser excesivo. El protocolo de comunicación procurará esto.
- **RNF2: Número de conexiones.** El servidor debe soportar múltiples conexiones de usuarios. Para ello el servidor escuchará en un puerto a los clientes que se quieran conectar.
- **RNF3: Eficiencia.** El servidor no debe saturarse debido a un número de conexiones no demasiado elevado.
- **RNF4: Simplicidad de uso.** La interfaz desarrollada para el chat no debe ser complicada de usar.

3.5. Diagrama de casos de uso y de flujo.

El sistema se compone de una actor principal que será el usuario que se quiere comunicar por el chat.

Este usuario puede realizar las siguientes funcionalidades:

- Acceder al chat.
- Enviar mensajes.
- Cambiar de nick.
- Desconectarse.

Todas estas funciones generaran procesos internos dentro del sistema que satisfarán al usuario. El diagrama de casos de uso se puede ver en la figura 3.7.

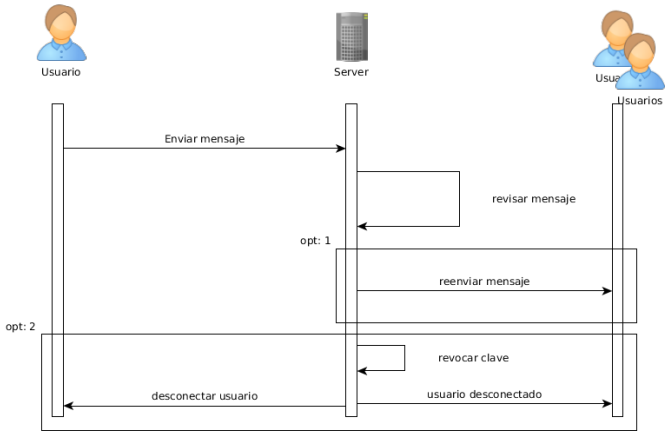


Figura 3.8: Diagrama de secuencia del proceso de envío de mensajes.

En general, todos los procesos internos del sistema son bastante sencillos y soló requieren recibir el mensaje, modificarlo un poco y reenviarlo a los usuarios. El único proceso más complejo es el que se realiza al recibir un mensaje de un usuario, este proceso puede acabar de dos formas: reenviando el mensaje a los usuarios o siendo revocada la clave que ha producido esa firma. El diagrama de secuencia de este proceso se puede ver en la figura 3.8.

4
Desarrollo

En este capítulo se detallan las herramientas usadas para desarrollar el protocolo de comunicación, el chat de usuarios y el sistema de revocación de claves propuestos en el capítulo 3. Las secciones de este capítulo ayudan a implementar los subsistemas propuestos en la sección 3.4. El subsistema de conexión se explica en las secciones 4.1 y 4.2, el subsistema de chat en las secciones, 4.3 y 4.4, el subsistema de revocación en la sección 4.5.

4.1. OpenSSL

OpnenSSL (OpensSSL, 2016) es una de las herramientas que se ha usado en este trabajo para desarrollar varios protocolos. Es un proyecto de código abierto con un fuerte soporte para muchos criptosistemas y el protocolo de comunicación TLS/SSL. Tiene un licencia similar a la de apache, que permite tanto un uso comercial como no comercial sin ningún tipo de restricción.

El programa principal de OpenSSL consiste en una línea de comandos con distintos protocolos que permiten cifrar, descifrar, generar certificados, etc. Es una herramienta muy potente ya que recoge un amplísimo número protocolos. Sin embargo, aunque hay ciertas partes de este programa que sí se han usado, como la generación de claves RSA, lo que más se ha usado de OpenSSL es una librería escrita para el lenguaje C que contiene la mayoría de estos protocolos. Esta librería se llama **libssl** (libssl, 2016) y en sucesivas secciones se detalla cómo se ha usado esta librería para crear las distintas partes del protocolo propuesto.

4.1.1. RSA para firmas

En esta sección se explica cómo se ha desarrollado la funcionalidad de firma de mensajes por parte del servidor. Antes de nada hay que explicar que es necesario generar las claves RSA del servidor para que funcione, ya que el programa espera las claves para poder firmar mensajes con ellas.

Para generar las claves RSA del servidor necesitaremos usar la herramienta de línea de comandos de OpneSSL. Usaremos, por tanto, los siguientes comandos:

```
openssl genrsa -out privkey.pem 2048
```

```
openssl rsa -in privkey.pem -pubout -out pubkey.pub
```

- **genrsa**: permite generar una nueva clave privada para RSA. Necesitamos especificar el tamaño de la clave, en este caso **2048** bits y opcionalmente el fichero de salida con **-out**, como se ve se ha especificado que se escriba en **privkey.pem**.
- **rsa**: esta opción, entre otras cosas, nos permite derivar la clave pública con la opción **-pubout**, para ello especificamos la clave privada **-in privkey.pem** y la salida **-out pubey.pem**

Una vez generadas las claves satisfactoriamente se han desarrollado unas funcionalidades para cargar estas claves, firmar mensajes y verificarlos. Antes de nada explicar que el modelo de firma que se usa es el explicado en la sección 2.2.1. Que usara SHA256 como función de hash.

En concreto estas funciones implementan `signRSAmsg` y `checkRSAsign` de la figura 3.3.

Las funciones desarrolladas se pueden separar en dos categorías cargar y guardar claves RSA y firmar y verificar mensajes. Las funciones de cargar y guardar son:

```
int RSAfileToPubKey(EVP_PKEY** pubKey, char* fname);
int RSApubKeyToFile(EVP_PKEY* pubKey, char* fname, int* msglen);
int RSAfileToPrivKey(EVP_PKEY** privKey, char* fname);
int RSAprivKeyToFile(EVP_PKEY* privKey, char* fname, int* msglen);
```

En resumen tenemos cuatro funciones, dos para cargar las claves pública y privada (**RSAfileToPubKey**, **RSAfileToPrivKey**), y dos para guardarlas (**RSApubKeyToFile**, **RSAprivKeyToFile**).

Las funciones de firma y verificación se encuentran en las siguientes funciones:

```
int signMsgRSA(EVP_PKEY* key, const unsigned char* msg,
    unsigned char** sig, size_t* slen, size_t msglen);

int verifySignRSA(EVP_PKEY* key, const unsigned char* sig,
    const unsigned char* msg, size_t slen, size_t msglen);
```

La función **signMsgRSA** genera la firma **sig** a partir de un mensaje **msg**, es necesario que se haya cargado previamente la clave privada **key**, que el valor **msg** no sea inválido y que pasemos la longitud de **msg** en el argumento **msglen**. En caso de que lo anterior se cumpla se generara la firma del mensaje **sig** y se nos devolver la longitud de está por el argumento **slen**.

La función **verifySignRSA** verifica si la firma **sig** se ha generado a partir del mensaje **msg**, para ello necesitamos tener la clave pública **key** y los valores **sig** y **msg** con sus respectivas longitudes **slen** y **msglen**.

A parte de estas funciones se han desarrollado otras, que aunque no tienen mucho objetivo para este proyecto han sido útiles para realizar pruebas. El código completo de las funciones implementadas se puede encontrar en el anexo N.3.

4.1.2. DH para intercambio de claves

El intercambio de claves mediante Diffie Hellman (ver anexo C) se ha desarrollado también usando la librería de OpensSSL. Estas funciones implementan `genDHkeys` y `deriveSharedK` de las figuras 3.3. y 3.4.

Encontramos aquí tres grupos de funciones distintas. Las que nos permiten generar claves:

```
int getParamsIniDH(EVP_PKEY** params);
int genNewParamsIniDH(EVP_PKEY** params, EVP_PKEY_CTX** pctx);
int genKeyFromParamsDH(EVP_PKEY_CTX** kctx, EVP_PKEY** dhkey,
    EVP_PKEY* params);
```

Las funciones **getParamsIniDH** y **genNewParamsIniDH** generan ambas las claves iniciales del intercambio **params**, la diferencia es que **getParamsIniDH** usa valores que ya nos aporta OpenSSL y que **genNewParamsIniDH** los genera desde cero.

Por otro lado **genKeyFromParamsDH** genera la clave complementaria **dhkey** a partir de la clave pública inicial **params**.

El siguiente grupo de funciones deriva la clave compartida y está compuesto por la función:

```
unsigned char* deriveSharedSecretDH(EVP_PKEY* privkey,
    EVP_PKEY* peerkey);
```

deriveSharedSecretDH recibe una las clave privada **prvkey** y la clave pública **peerkey** y deriva el secreto compartido.

El último grupo de funciones sirve para generar un cadena de texto a partir de la clave, recibiendo la clave privada y escribiendo la pública. Las funciones de este grupo son:

```
int DHpubKeyToMsg(EVP_PKEY* pubKey, char ** msg);
int msgToDHpubKey(EVP_PKEY** pubKey, char * msg, int msglen);
```

DHpubKeyToMsg genera un la cadena **msg** de texto a partir de la clave **pubKey** y nos devuelve la longuitud de ésta.

msgToDHpubKey genera la clave pública **pubkey** a partir de la línea de texto **msg** y su respectiva longitud **msglen**.

Los códigos de estas funciones se pueden ver en el anexo N.2.

4.1.3. AES CBC

El cifrado de mensajes se lleva a cabo mediante AES 256 en el modo cbc (ver anexo D). Las funciones desarrolladas nos permitirán transmitir información de forma privada.

Las funciones desarrolladas para este módulo se dividen en crear el contexto de cifrado y cifrar y descifrar. Estas funciones implementan `AES` de las figuras 3.5 y 3.6.

La función para inicializar el contexto es:

```
EVP_CIPHER_CTX* create_ctx();
```

Y las funciones para cifrar y descifrar son:

```
int encrypt_cbc256(EVP_CIPHER_CTX* ctx, unsigned char* key,
    unsigned char* iv, const unsigned char* text,
    unsigned char** out, int textlen);

int decrypt_cbc256(EVP_CIPHER_CTX* ctx, unsigned char* key,
    unsigned char* iv, const unsigned char* text,
    unsigned char** out, int textlen);
```

La función **encrypt_cbc256** encripta el texto **text**, con su respectiva longitud **textlen** usando el contexto **ctx**, la clave simétrica **key** y el vector de inicialización **iv**, produciendo el texto cifrado **out** y nos devolverá el valor de este texto.

La función **decrypt_cbc256** desencripta el mensaje **text** de longitud **textlen** usando el contexto **ctx**, la clave simétrica **key** y el vector de inicialización **iv**, produciendo en caso satisfactorio **out** y nos devolverá el tamaño de este texto.

Los códigos de estas funciones se encuentran en el anexo N.1

4.2. Libgrupsig

Libgroupsig, (libgrupsig, 2016), es una librería de firmas grupales que implementa los protocolos **CPY06**, **BBS04** y **KTY04**. Dentro de la librería hay programas que permiten probar el funcionamiento de las firmas, estos programas son. **claim**, **claim_verify**, **group_create**, **join**, **print_key**, **revoke**, **same_signer**, **sign**, **trace** y **verify**.

Estos programas funcionan para todos los protocoles salvo, **claim** y **claim_verify** que sólo funcionan para los protocolos **KTY04** y **CPY06** que están pensados para esta funcionalidad.

Todas las funciones creadas con esta librería desarrollarán las funciones `signGSmsg`, `verfyGSsign`, `traceGSsign` y `Sign` de las figuras 3.4 y 3.6 y permitirán la revocación explicada en la sección 3.3. Todo el código de estas funciones se encuentra en el anexo N.4.

4.2.1. Creación de grupos

La creación de claves se realiza mediante el programa de la librería **group_create**. Este programa establece las condiciones iniciales del grupo creando los directorios y los archivos:

- El directorio de miembros.
- El directorio del administrador.
- El directorio del grupo.
- El archivo gml. Este archivo contiene información sobre las claves de los miembros.
- El archivo crl. Este archivo contiene información sobre las claves revocadas.
- El archivo mgr.key. Es la clave del administrador del grupo.
- El archivo grp.key. Es la clave grupal.

Estos archivos guardan la información del grupo y permiten crear las claves de los miembros y otras funcionalidades que requieran el uso de la clave del administrador. Un ejemplo de ejecución para el esquema **CPY06** seria:

```
group_create CPY06 -d gs -M manager -g group -m member
```

- **Esquema**, en este caso este valor es **CPY06**.
- **-d gs** indica que el directorio del grupo es **gs**.
- **-M manager** indica que el directorio del administrador del grupo es **manager**.
- **-g group** indica que el directorio de la clave de grupo es **gs**.
- **-m member** indica que el directorio de las claves de miembro es **members**.

4.2.2. Añadir claves

Para crear claves existe el programa **join** que nos permite añadir el número de claves que especifiquemos al grupo.

```
join CPYO6 grp.key mgr.key gml members 3
```

- **Esquema**, esquema de firmas grupales que se va a usar para generar las claves, en este caso **CPY06**.
- **grp.key**, la clave del grupo.
- **mgr.key**, la clave del administrador.
- **gml**, el listado de miembros.
- **3**, la cantidad de claves a crear.

Adicionalmente hace falta generar un archivo **crl** donde se guardará la identidad de los usuarios revocados.

La estructura de los archivos tras crear los grupos y añadir las claves será:

```
./fg/
 group
     grp.key
 manager
     crl
     gml
     mgr.key
 members
       0.key
       1.key
       2.key
```

Una vez creadas estas claves se reparten a los miembros del grupo[1].

4.2.3. Firma de mensajes

Para la firma de mensajes se ha desarrollado un código inspirado en el programa **sign** de la librería que dado un mensaje **msgstr**, la clave del miembro **memkey** y la del grupo **grpkey** produce la firma de ese mensaje **sigstr** y devuelve el tamaño de ésta.

```
int signMsgGS(groupsig_key_t* grpkey, groupsig_key_t* memkey,
uint8_t scheme, char *msgstr, char** sigstr);
```

4.2.4. Verificación de mensajes

La verificación de mensajes también está inspirada en un programa de la libreria, en este caso en el programa **verify**. Para desarrollar esta funcionalida se ha implementado una función tal que dado un mensaje **msgstr**, una firma **sigstr** y la clave del grupo **grpkey**, comprueba que la firma es válida para ese mensaje.

[1]Un problema interesante, no tratado en este trabajo, sería como repartir estas claves de forma segura.

```
int verifySignGS(char* sigstr, groupsig_key_t *grpkey,
char* msgstr, uint8_t scheme);
```

4.2.5. Trazabilidad de firmas válidas

La trazabilidad de firmas es un proceso que comprueba si una una firma ha sido producida con una clave revocada. Para ello se usa una función de la librería que lo comprueba basada en el programa **trace**. Esta operación solo se realiza es el proceso de handshake. Se ha desarrollado para ello una función que comprueba que la clave no este revocada.

```
int traceSignGS(char* sigstr, groupsig_key_t *grpkey,
groupsig_key_t *mgrkey, crl_t* crl, gml_t* gml, uint8_t scheme);
```

4.2.6. Revocación de firmas

La revocación consiste en revocar una de las claves de los miembros para que no se pueda volver a conectar al grupo, ya que será detectado cuando se haga un trace sobre una firma. La función que implementa esta capacidad está basada en el programa **revoke**.

```
int revokeSigGS(groupsig_signature_t *sig, groupsig_key_t *grpkey,
groupsig_key_t *mgrkey, gml_t *gml,  crl_t *crl,
uint8_t scheme, char *s_crl);
```

Esta función recibe la firma **sig**, la firma del grupo y la del administrador **grpkey mgrkey**, así como **gml** y **crl** y por último el lugar donde se encuentra el archivo el crl **s_crl** para poder actualizarlo.

4.3. Chat

El chat desarrollado para este trabajo se compone de una serie de clientes y un servidor que les permite comunicarse entre ellos. Para la comunicación entre ambos se han desarrollado una serie funciones que permiten enviar mensajes, recibirlos e iniciar el proceso de conexión mediante las firmas grupales.

4.3.1. sconexion

sconexion es una interfaz para el intercambio de mensajes entre el cliente y servidor mediante los protocolos descritos en el diseño, secciones 3.1.1 y 3.1.2. Está compuesta por las distintas funciones para la comunicación y una estructura **Sconexion_t** con los valores necesarios para este tipo de conexión. Estos valores son distintos en caso de que sea el cliente el que los use o el servidor, ya que el cliente si tendrá clave de miembro, mientras que el servidor no la requerirá.

Los códigos de estas funciones puede ser consultado en el anexo N.5.

4.3.2. Cliente y servidor

Cliente y servidor tienen una serie de comandos y procesos para intercambiar y validar información entre ellos.

Un servidor estará a la espera de que se inicien conexiones con los clientes y una vez iniciada la conexión con un cliente y comprobado que es un miembro del grupo, el servidor espera a que el cliente envíe su nick para informar que un nuevo miembro se ha conectado y creará un hilo de escucha para ese cliente.

El servidor guardará una lista con los clientes que están conectados, por simplicidad ésta es un lista enlazada, que guarda la información del cliente como el nick, y la información de la conexión con el cliente, claves, etc.

Para la comunicación entre ambos el cliente tendrá un canal de escritura y un canal escucha, mientras que el servidor solo tendrá un canal de escucha por cada cliente. Los canales de escucha de ambos reaccionaran de distinto modo en función del comando que reciban. Y el canal de escritura del cliente esperará que el usuario escriba un cierto comando para enviárselo al servidor.

Por otro lado el servidor tiene un hilo para comprobar que las conexiones con los clientes no se han perdido. En caso de que una conexión con un cliente se pierda se informará al resto de clientes que ese cliente se ha desconectado.

El código de esta parte se encuentra en el anexo N.6.

4.4. ncurses

ncurses, (ncurses, 2016), es un librería para C que implementa una interfaz de usuario por terminal. Entre otras cosas, ncurses tiene la capacidad de crear ventanas, estas ventanas son cuadros de distintas formas que ocupan un cierto espacio. Entre las funcionalidades que tienen estas ventanas se encuentran la de escribir, leer de ellas, moverlas, etc.

La interfaz que se ha desarrollado estará integrada en el programa del cliente haciendo más cómoda para el usuario la recepción y envió de mensajes. Para ello se han creado dos ventanas. Una de las ventanas tendrá la función de mostrar al usuario los mensajes recibidos y la otra ventana escanear la entrada de mensajes por parte del usuario. Estas dos ventanas se adaptarán al tamaño de la terminal donde se esté ejecutando el programa.

El código de esta interfaz puede consultarse en el anexo N.7. Y el aspecto del programa cliente se puede ver en el anexo M

4.5. Revocación

El sistema de revocación que se ha implementado es un sistema bastante sencillo. Consiste principalmente en una lista de palabras que no se pueden decir en el grupo y en caso de que usen se pasará a usar las funciones de revocación para revocar la clave de ese miembro. La función para detectar si un mensaje contraviene las normas del grupo es:

```
int bigBrother(char* msg, int msglen);
```

Básicamente devuelve **1** si el mensaje **msg** contraviene las normas y **0** en caso contrario. El código de estas funciones se encuentra en el anexo N.8.

5

Pruebas y resultados

En este capítulo se llevan a cabo pruebas del protoclo propuesto relativas a la eficiencia computacional en la sección 5.1 y el tamaño de los mensajes en la sección 5.2.

5.1. Pruebas de eficiencia

Las siguientes pruebas se han realizado en un procesador intel core i3 5005U de 2.00 Ghz con 2 núcleos (4 usando hypetrheading) y 3MB de cache. El esquema de firmas que se ha usado para estas pruebas es CPY06.

5.1.1. Tiempo medio de inicio de conexión

En esta sección se ha realizado una prueba para comprobar cuántos ciclos de reloj consume el establecimiento de la conexión tanto del cliente como del servidor. Para ello se ha realizado una prueba en la que se mide el tiempo consumido para establecer la conexión con 50 clientes, tanto por parte del cliente como por parte del servidor. La figura 5.1 muestra los resultados obtenidos.

Como se observa el tiempo para establecer la conexión es mayor por parte del servidor que por parte del cliente. Esto se debe a que, como se comentó en el cuadro 2.1, el proceso de verificación en el esquema CPY06 es más costoso que el proceso de firma, y al realizarse esta operación dos veces, una para el intercambio de la clave y otra para el intercambio del vector de inicialización, el servidor requiere de más computó que el cliente. Como se refleja en el tiempo medio de establecimiento de conexión, que es de 143.743 ciclos de reloj para el servidor y de 96.686 ciclos de reloj para el cliente.

5.1.2. Tiempo de envío de mensajes

En esta sección se muestran los resultados obtenidos para el envío de mensajes de cliente y servidor. El tiempo de envío será el tiempo que se tarda en generar el mensaje y no el tiempo de recepción.

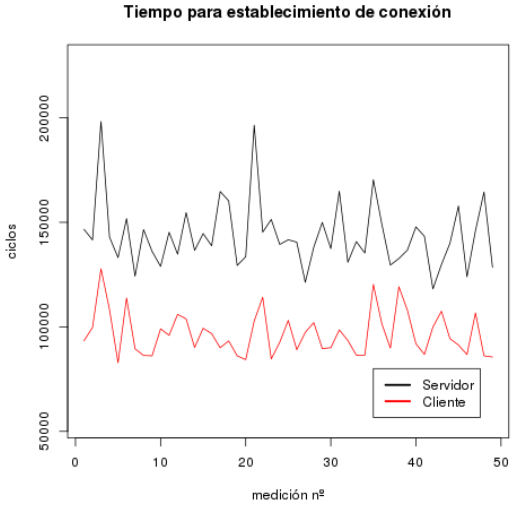


Figura 5.1: Tiempo para el establecimiento de la conexión de cliente y servidor.

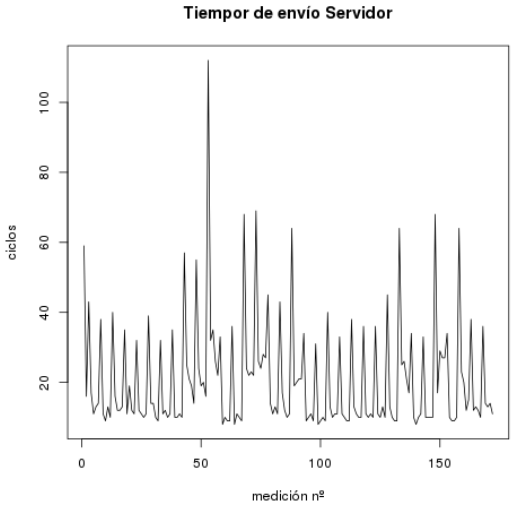


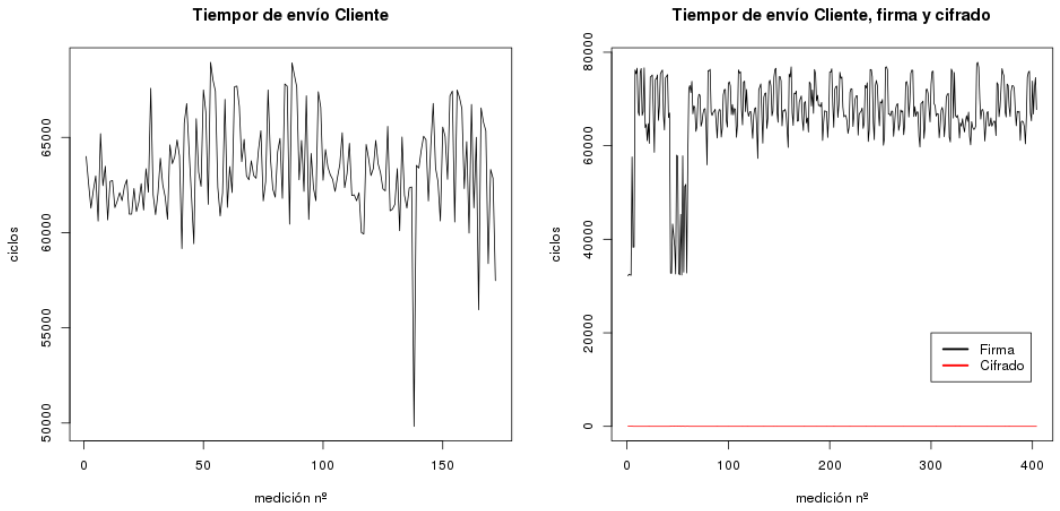
Figura 5.2: Envío de mensajes por el servidor.

Servidor

En la figura 5.2 se puede ver el tiempo que tarda el servidor en realizar este proceso. Esta prueba ha consistido en generar varios mensajes para cinco clientes distintos y medir el tiempo que se requiere desde generarlo hasta enviarlo. Como se observa es bastante rápido, esto se debe a que el servidor únicamente tiene que cifrar el mensaje. El tiempo medio obtenido de esta prueba ha sido de 20,48256 ciclos de reloj.

Cliente

En la figura 5.3a se muestran los resultados obtenidos para el envió de mensajes desde el cliente. Esta prueba es análoga a la prueba de tiempos de envío del servidor, solo que del lado del cliente. Como se puede observar el tiempo requerido para generar un mensaje por el cliente es mucho mayor que el del servidor esto se debe a que el proceso para generar la firma es mucho más costoso que el de cifrado. El tiempo medio obtenido para esta prueba 63357,45 ciclos de



(a) Tiempo total.

(b) Firma y cifrado.

Figura 5.3: Envío de mensajes por el cliente.

reloj.

En la figura 5.3b se puede ver el tiempo que requiere para el cliente firma un mensaje y el tiempo que requiere cifrar un mensaje. Como se observa hay una gran diferencia entre un proceso y otro obteniendo un tiempo medio de 66870,67 ciclos de reloj para la firma y un tiempo medio 8,423267 de ciclos de reloj para el cifrado.

5.2. Tamaño de los mensajes

Un factor importante de las comunicaciones de red es saber cuánto ocupan los datagramas que circulan por ella, siendo un aspecto importante el tamaño de la información que se genera. Para medir esto se ha realizado una prueba sobre cuánto ocupa un mensaje firmado.

Más que una prueba, los resultados obtenidos son mediciones prácticas, ya que el tamaño de un mensaje seguirá una fórmula matemática muy sencilla compuesta por el tamaño del mensaje a enviar y el tamaño de la firma, siguiendo la fórmula $T_{total} = T_{firma} + T_{men}$. En la figura 5.4 se observa el tanto por ciento que ocupa una firma grupal usando el esquema CPY06 sobre un mensaje firmado. Si se compara éste tipo de firmas con otro, como por ejemplo un firma RSA de 2048 bits que se supone estándar (NIST, 2016), se ve que el tamaño de la firma producida es parecido, ya que RSA genera unas firmas de 256 bytes y CPY06 de 362 bytes.

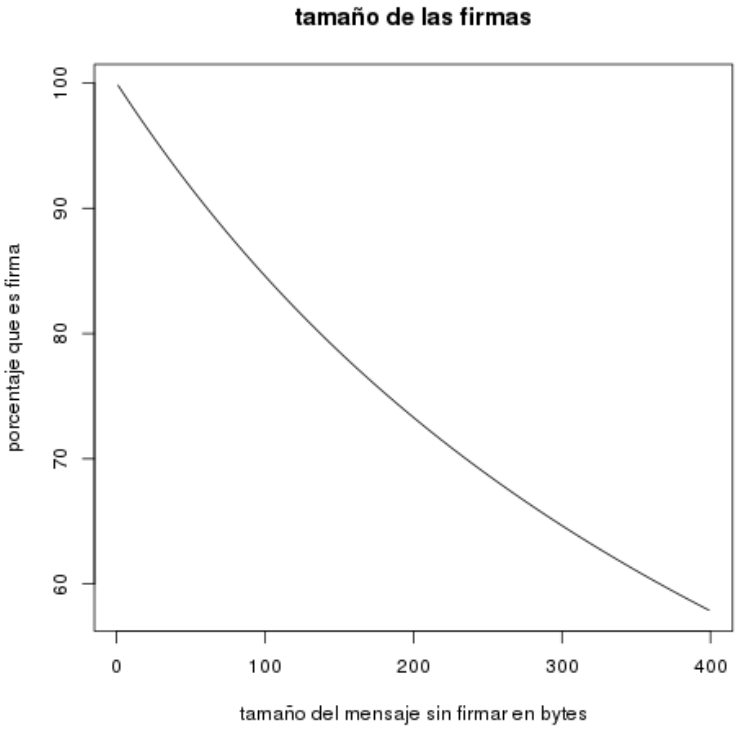


Figura 5.4: Tanto por ciento de un mensaje ocupado por una firma grupal.

6

Conclusiones y trabajo futuro

6.1. Conclusiones

En este trabajo se ha estudiado y analizado el concepto de firma grupal, en concreto los sistemas que hacen uso de firmas revocables y firmas cortas, y se ha construido un protocolo de comunicación seguro, anónimo y justo, es decir revocable, que hace uso de la firma grupal como método de verificación de identidad.

Con carácter previo se han descrito los sistemas existentes de firma digital, como la firma mediante RSA, y la relación de la firma digital con otros protocolos criptográficos para conseguir comunicaciones seguras.

A continuación se han analizado tres esquemas de firma grupal: **KTY04**, **CPY06** y **BBS04**, que hacen uso de las pruebas de conocimiento cero.

Una vez desarrolladas las bases teóricas de este trabajo se ha pasado realizar una aplicación práctica mediante el diseño de un protocolo de comunicación que hace uso de las firmas grupales como método de identificación de miembros de un grupo anónimo.

El protocolo diseñado se ha probado creando un chat de comunicación de internet y ha demostrado su correcto funcionamiento. Como conclusiones practicas se ha comprobado que el protocolo no sobrecarga excesivamente la red en comparación con firmas RSA de 2048 bits, aunque requiere mucho más computo debido a que cada mensaje enviado está firmado para permitir la revocación, a diferencia de otros sistemas. Una posible mejora sería eliminar el proceso de firmado que se realiza en cada mensaje y solo usar el protocolo para crear un canal anónimo.

Por todo ello se concluye que la firma grupal es válida para protocolos de comunicación anónimos privados y justos.

6.2. Trabajo futuro

Como trabajo futuro sería interesante profundizar en los siguientes aspectos:

- Certificar y comprobar la clave pública del servidor.

- Realizar pruebas de seguridad exigentes mediante alguna metodología (Diaz et al., 2014a). Identificando las posibles fugas del protocolo y los puntos unicos de fallo.
- Añadir la posibilidad de usar otros protocolos criptográficos en las distintas partes del intercambio o del canal de comunicación.
- Idear un mecanismo para la distribución de las claves de miembros. En este trabajo se está asumiendo que cada cliente tiene ya su propia clave, pero no existe un sistema formal para la distribución de estas. Por tanto debería ser prioritario idear un sistema de este tipo.
- Realizar una publicación científica sobre métodos de comunicación que hagan uso las firmas grupales en una revista especialidad.

Bibliografía

BOE298 (1999). Boe298. [Online; accessed 6-December-2016].

Boneh, D. (1998). *The Decision Diffie-Hellman problem*, pages 48–63. Springer Berlin Heidelberg, Berlin, Heidelberg.

Boneh, D., Boyen, X., and Shacham, H. (2004). *Short Group Signatures*, pages 41–55. Springer Berlin Heidelberg, Berlin, Heidelberg.

Chaum, D. (1983). *Blind Signatures for Untraceable Payments*, pages 199–203. Springer US, Boston, MA.

Chaum, D. and van Heyst, E. (1991). *Group Signatures*, pages 257–265. Springer Berlin Heidelberg, Berlin, Heidelberg.

Choi, S. G., Park, K., and Yung, M. (2006). *Short Traceable Signatures Based on Bilinear Pairings*, pages 88–103. Springer Berlin Heidelberg, Berlin, Heidelberg.

Diaz, J., Arroyo, D., and Rodriguez, F. B. (2014a). A formal methodology for integral security design and verification of network protocols. *Journal of Systems and Software*, 89:87 – 98.

Diaz, J., Arroyo, D., and Rodriguez, F. B. (2014b). libgroupsig: An extensible c library for group signatures. .

Diffie, W. and Hellman, M. (2006). New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654.

Ehrsam, W., Meyer, C., Smith, J., and Tuchman, W. (1978). Message verification and transmission error detection by block chaining. US Patent 4,074,066.

Kiayias, A., Tsiounis, Y., and Yung, M. (2004). *Traceable Signatures*, pages 571–589. Springer Berlin Heidelberg, Berlin, Heidelberg.

libgrupsig (2016). libgrupsig. [Online; accessed 6-December-2016].

libssl (2016). libssl. [Online; accessed 6-December-2016].

Merkle, R. and Hellman, M. (1978). Hiding information and signatures in trapdoor knapsacks. *IEEE transactions on Information Theory*, 24(5):525–530.

ncurses (2016). ncurses. [Online; accessed 6-December-2016].

NIST (2001). Announcing the advanced encryption standard (aes). [Online; accessed 6-December-2016].

NIST (2016). keylenght recomendation. [Online; accessed 6-December-2016].

OpensSSL (2016). Opensssl. [Online; accessed 6-December-2016].

Pfitzmann, A. and Hansen, M. (2009). A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management.

Prof. Dr. Mark Manulis, Nils Fleischhacker, F. G. F. K. B. P. (2012). *Group Signatures: Authentication with Privacy*. Cryptographic Protocols Group Department of Computer Science Technische Universität Darmstadt, Mornewegstrasse 30 64293 Darmstadt GERMANY.

Quisquater, J.-J., Guillou, L., Annick, M., and Berson, T. (1989). How to explain zero-knowledge protocols to your children. In *Proceedings on Advances in Cryptology*, CRYPTO '89, pages 628–631, New York, NY, USA. Springer-Verlag New York, Inc.

rfc2818 (2016). rfc2818. [Online; accessed 6-December-2016].

rfc5246 (2016). rfc5246. [Online; accessed 6-December-2016].

rfc5280 (2016). rfc5280. [Online; accessed 6-December-2016].

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126.

Rivest, R. L., Shamir, A., and Tauman, Y. (2001). *How to Leak a Secret*, pages 552–565. Springer Berlin Heidelberg, Berlin, Heidelberg.

Stallings, W. (2014). *Cryptography and network security: Principles and practice.* Pearson, Boston, sixth edition edition.

tor (2016). Tor project. [Online; accessed 6-December-2016].

A

Conocimientos adquiridos

La realización de este trabajo me ha ayudado a adquirir los siguientes conocimientos:

- Funcionamiento en detalle de los actuales protocolos de firma y de los elemntos de PKI.
- Funcionamiento de SSL/TLS.
- Funcionamiento de las firmas ciegas.
- Funcionamiento de las firmas de anillo.
- Funcionamiento de suposiciones computaciones complejas basadas en los problemas de Diffie-Hellman, como Decisional Diffie-Hellman o Linear Diffie-Hellman.
- Funcionamiento de las pruebas de conocimiento cero.
- Funcionamiento de las firmas digitales, tanto de los primeros modelos como de los modelos más avanzados como la firma grupal.
- Funcionamiento de la librería libssl.
- Funcionamiento de la librería libgruopsig.
- Funcionamiento de ncurses.
- Funcionamiento de las publicaciones científicas, al menos como leerlas y buscarlas.

He de decir que ha sido un trabajo entretenido de realizar, aunque muy extenso, y me ha aportado muchos conocimientos que espero que me sirvan en un futuro y que es una tarea interesante seguir al día de estos sistemas por el potencial que, al menos yo, consideró que tienen.

B

RSA.

El criptosistema RSA es un método criptográfico propuesto en (Rivest et al., 1978). Este sistema seguía la idea de sistemas anteriores (Diffie and Hellman, 2006) de cifrar un mensaje con una clave y descifrar con otra.

El funcionamiento de este sistema es bastante sencillo. Se necesitan escoger dos números primos p y q. Con estos dos números generamos un valor N tal que $N = pq$, este valor se usara como módulo. A continuación se computa el valor de la función de Euler para n, es decir, $\varphi(n) = n(1 - \frac{1}{q})(1 - \frac{1}{p})$ y conservamos este valor de forma privada. Elegimos un valor e tal que $q < e < \varphi(N)$ y tal que $gcd(e, \varphi(N)) = 1$. Por ultimo obtenemos el valor multiplicativo inverso de e que sera d, es decir, $d = e^{-1}\ mod\ \varphi(N)$. Con esto ya tenemos toda para cifrar y descifrar.

Para cifrar y descifrar podemos usar e o d de la siguiente forma:

- Cifrado: $c \equiv m^e mod\ N$.
- Descifrado: $c^d \equiv (m^e)^d mod\ N \equiv m$.

Para usar d para cifrar habría que sustituir las e por las d y viceversa.

C

Intercambio de claves mediante Diffie-Hellman

El intercambio de claves mediante Diffie-Hellman es un método de intercambio de información que depende de las suposiciones de complejidad del logaritmo discreto, explicado en el anexo F. Es un sistema que se usa para generar una clave común entre dos partes E y D que solo puede ser conocida por ellos. Este sistema es seguro en el supuesto de que una tercera parte F solo pueda ver el tráfico pero no pueda modificarlo.

El funcionamiento consiste en que E y D dan a conocer un numero q, que es primo, y una base b que es raíz primitiva de q, es indistinto quien los genere mientras sea una de los dos. Ocurre entonces lo siguiente:

1. E y D generan un x_e y un x_d.
2. Una vez teniendo estos tres números (x, b ,q) ambos pueden generar sus respectivo y_e y y_d según la formula $y = b^x \ mod \ q$.
3. Dan a conocer entonces sus respectiva y con las que la otra parte podrá conocer ya toda la clave pública (y,q y b).
4. Conocidas ya las claves públicas por ambas partes cada uno computa la calve común K usando la y de la otra parte como base y su x como exponente. E generaría así la clave $K = y_d^{x_e}$.

La demostración de porque este intercambio genera una clave común ambos es como sigue:

$K = y_d^{x_e} \ mod \ q = b^{x_d x_e} = b^{x_e x_d} = y_e^{x_d} \ mod \ q = K$.

D

Funcionamiento de AES en modo CBC

D.1. AES

AES (Advanced Encryption Standard) es un sistema de cifrado por bloques adoptado como estándar de encriptado por el NIST (National Institute of Satandards and Technology) en 2001 (NIST, 2001) después de que se empezase a poner en duda la seguridad que ofrecía el, por aquel entonces, algoritmo estándar de cifrado, DES.

Es un algoritmo que dado un bloque de texto y una contraseña cifra un mensaje. Para descifrar y cifrar se usa la misma operación, es decir la función inversa del encriptado es el mismo encriptado.

Es un algoritmo bastante complejo de entender y por ello no se va entrar a analizar aquí su corrección o su seguridad, sino que se a analiza uno de sus modos de funcionamiento más comunes, el modo CBC.

D.2. CBC

El modo Cipher Block Chaining (CBC) (Ehrsam et al., 1978), es un modo de operación para cifrados de bloques que realiza una XOR del texto inicial con un vector de inicialización y posteriormente cifra el resultado. Es un modo que funciona por rondas, para la primera ronda se usa un vector de inicialización y en posteriores rondas se usa el anterior texto cifrado como vector de inicialización. En la figura D.1 se puede ver su funcionamiento.

Para descifrar se hace la operación XOR después de descifrar con AES, obteniendo el mensaje original. En la figura D.2 se puede ver el funcionamiento.

Éste modo se usa frente a otros por varios motivos: es bastante eficiente ya que, a pesar de que el encriptado es secuencial, el desencriptado es paralelizable ya que se cuenta con todos los bloques a desencriptar. Es bastante seguro ya que solo depende de cómo se genere el vector de inicialización y la clave que se use y al usar partes anteriores del cifrado como entrada una sección de información que tenga el mismo contenido que otra no produce el mismo resultado, lo que aporta mayor seguridad.

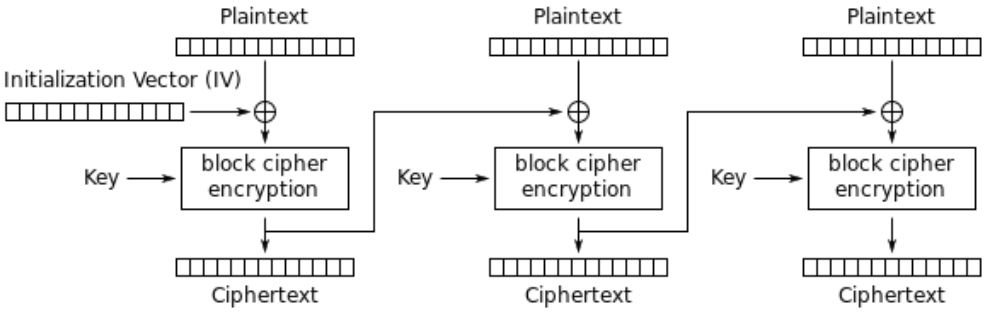


Figura D.1: CBC en modo cifrado.

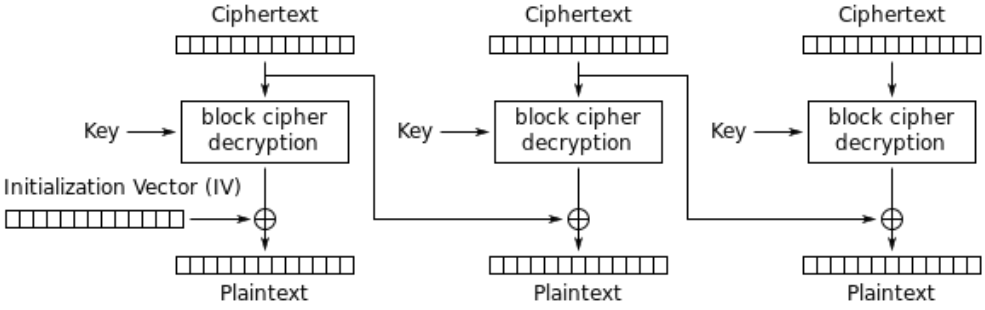


Figura D.2: CBC en modo descifrado.

E

Descomposición de un entero en sus factores primos.

Supongamos que tenemos un número N formado por el producto de dos números primos p y q, es decir:

$$pq = N \tag{E.1}$$

El problema de factorizar un número consiste en conocer que números, p y q, son los que forman N. A pesar de que es un problema bastante sencillo, computacionalmente es un problema muy costoso.

Un algoritmo sencillo e ilustrativo de este problema, aunque bastante malo, consistiría en lo siguiente:

1. Dividimos N entre 2 y nos da como resultado un entero n.
2. Probamos las distintas combinaciones de productos de los números que hay entre 2 y n hasta que obtenemos como resultado N.

Esta opción es bastante mala y muy ineficiente ya que tenemos que hacer un elevadísimo número de operaciones para obtener el resultado.

Todos los algoritmos conocidos resuelven este problema en tiempo exponencial y es por ello que se usa en criptografía. Sin embargo el algoritmo de Shor es capaz de resolverlo en tiempo polinómico, pero su principal inconveniente es el estado de la computación cuántica en la actualidad, ya que no permite realizar operaciones de números cuánticos de más de unos q-bits.

Strong-RSA

Strong-RSA es una suposición computacional basada en la factorización de un número. Consiste en que dado un número compuesto n y un $z \in QR(n)$, siendo $QR(N)$ el residuo cuadrático de n, es inviable computacionalmente encontrar un $u \in \mathbb{Z}_n^*$ y un $e > 1$ tal que $u^e = z(\ mod\ n)$.

F

Logaritmo discreto.

El logaritmo discreto es una función matemática que permite calcular dados una base b, un valor de modulo n y un valor cualquiera y su correspondiente x.

$$\log_b y \ mod \ n = x$$

Por las propiedades de los logaritmos esta función se puede expresar de manera exponencial:

$$y = b^x \ mod \ n$$

El uso de estas funciones en la criptografía es muy útil y se usa en multitud de protocolos criptográficos, como por ejemplo Diffie-Hellman o el ElGamal. Sin embargo, falta añadir unas cuantas cosas más para entender porque es seguro usar el logaritmo discreto, para ello hay que explicar que es el teorema de Euler y que son los primos relativos, ya que el teorema de Euler hace uso de ellos.

Un número a es primo relativo a otro b si entre ellos no tienen factores divisibles comunes, es decir si el MCD de ambos números es 1.

Para un cierto número n si le aplicamos la función de Euler ϕ obtenemos que cantidad de primos relativos hay.

Por otra parte lo que indica el teorema de Euler es que dados a y n que son coprimos estos tienen la siguiente propiedad:

$$a^{\phi(n)} = 1 \ mod \ n$$

Sin embargo, hay otros valores distintos de $\phi(n)$ que cumplen que,

$$a^b = 1 \ mod \ n$$

donde b es un valor entre 1 y $\phi(n)$.

Una vez entendido el teorema de Euler y sus propiedades volvemos a los logaritmos discretos. Estos son interesantes ya que en la formula anterior, que es la del logaritmo discreto en forma de una exponencial, se puede variar a y b con respecto a n obteniendo las siguientes propiedades:

a	a^1	a^2	a^3	a^4
$a = 1$	1	1	1	1
$a = 2$	2	4	3	1
$a = 3$	3	4	2	1
$a = 4$	4	1	4	1

Cuadro F.1: Potencias de modulo 5.

1. Las secuencias para cualquier a es cíclica y termina en uno.
2. algunas secuencias son de longitud $\phi(n)$. Si esto ocurre se dice que a es raíz primitiva de n.

Un ejemplo se puede observar en la tabla F.1 donde se muestran las potencias de módulo 5.

Si usamos un número primo como n y otro número a, que sea raíz primitiva, obtenemos un conjunto de enteros entre 1 y $n - 1$ que no se repiten, por tanto al operar conseguimos otro número que es difícil averiguar cómo se origina si no se conocen las condiciones previas del problema.

Este sistema es computacionalmente seguro ya que la seguridad reside en conocer la x de la ecuación $y = b^x \ mod \ n$. En todos los criptosistema que dependen del logaritmo discreto, el exponente x solo lo conoce la parte que lo ha generado e intentar obtenerlo conociendo el resto de factores, es decir, resolviendo la ecuación $x = \log_a(y) \ mod \ n$ es computacionalmente inviable.

Este problema tiene una magnitud de complejidad parecida a factorizar los primos de un entero, para ambos existen soluciones aplicables al otro y existen algoritmos cuánticos capaces de resolverlo en tiempo polinómico.

G

Firmas ciegas

Las firma ciegas son un protocolo de firma propuesto en (Chaum, 1983), en el que un mensaje se oculta antes de ser firmado. Este tipo de sistemas se puede usar en protocolos de voto y dinero electrónico.

En un sistema de firma ciega están implicados una entidad que firma S una persona que crea un mensaje A y una persona que comprueba que el mensaje ha sido firmado B.

S tendrá las claves pública S_{pub} y privada S_{priv}. A tendrá las claves pública C_{pub} y privada C_{priv}. Además existirá una función r que se usara para ofuscar el mensaje original.

El proceso para generar la firma consiste en:

1. A hace pasar el mensaje x por $r(x)$ para comprobar la redundancia del mensaje, se aplica $C_{pub}(x)$ y A envía el resultado a S.
2. S aplica $S_{priv}(C_{pub}(x))$ y devuelve el resultado a A.
3. A aplica su propia clave privada sobre el mensaje, $C_{priv}(S_{priv}(C_{pub}(x))) = S_{priv}(x)$ y envía el resultado a B.
4. B aplica la clave pública de S para descubrir el mensaje, $S_{pub}(S_{priv}(x)) = x$

Este sistema es fácilmente implementable con un protocolo de clave asimétrica como RSA.

Tenemos los actores que hemos comentado antes, la función r consiste en hacer el producto de un valor aleatorio r con m. El exponente público de S es e_s, el privado d_s, los exponentes de A son el valor público e_a y el valor privado d_a

1. A aplica $m' = (mr)^{d_c} \ mod \ N$ que oculta el mensaje inicial y envía el resultado a S.
2. S aplica $s' = (m')^{e_s} \ mod \ N$, que firma el mensaje y devuelve el resultado a A.
3. A resuelve $s = (s')^{e_c} r^{-1} \ mod \ N = (mr)^{d_c e_s e_c} r^{-1} \ mod \ N = m^{e_s} \ mod \ N$ y envía el resultado a B.
4. B aplica $s^{d_s} \ mod \ N = m^{e_s d_s} \ mod \ N = m$ obteniendo el mensaje.

H

Firmas de anillo

La firma de anillo es un tipo de firma grupal en la que una serie de miembros puede producir una firma que verifique que ha sido generada por uno de los miembros, siendo imposible saber el miembro concreto que la genero.

Este tipo de firma digital se propuso por primera vez en (Rivest et al., 2001) inspirado por anteriores trabajos sobre firmas grupales.

Esta firma se diferencian de otras firmas grupales en que no necesitan una autoridad central que emita las claves, no hay un procedimiento inicial ligado al grupo para crear las claves que se van a usar y no es necesario el consentimiento o conocimiento del resto de miembros para producir un mensaje. El único requerimiento es que el firmante use algún algoritmo de clave pública y que una serie de usuarios hayan publicado sus claves públicas, que se usaran para producir la firma en cuestión. La idea es que los usuarios interesados en este tipo de sistema publiquen su clave pública en algún directorio público.

Para que un miembro del grupo pueda generar una firma σ sobre un mensaje únicamente necesita el mensaje m, su clave secreta PR_s y la clave pública de los miembros del grupo PU_1, PU_2, ..., PU_r. Y para verificar que el mensaje proviene del grupo se necesitan el mensaje m, la firma σ y la lista claves públicas de los miembros del grupo ω.

Funcionamiento

Los elementos de un sistema de anillo son un grupo G compuesto por los miembros A_1, A_2, ..., A_r, un miembro de este grupo A_s, $1 \leq s \leq r$ que quiere enviar un mensaje m y un verificador B que quiere comprobar que el mensaje proviene de G.

Cada miembro del anillo A_i tiene una función pública $f_i(x)$ y una función privada $f_i^{-1}(x)$. Típicamente estas funciones hacen uso de un sistema de clave pública/privada como RSA, por tanto la función pública consistirá en encriptar con la clave pública y la función privada encriptar con la clave privada. Estas funciones se pasan a un dominio en el que produzcan un valor de la misma longitud que las funciones de los otros miembros, por tanto cada usuario tendrá las funciones pública y privada $g_i(x)$ y $g_i^{-1}(x)$..

$$g(m) = \begin{cases} q_i n_i + f_i(r_i) & \text{si } (q_i + 1)n_i \leq 2^b \\ m & \text{sino} \end{cases}$$

Además de las funciones $g_i()$ y $g_i^{-1}()$ existe una función de cifrado simétrico $E_k()$ con su correspondiente función de descifrado $E_k^{-1}()$ y por ultimo una función hash $h()$ que produce una código de tamaño l que se usara como clave simétrica k.

El funcionamiento de estas firmas se ilustra en la figura H.1 y se divide en firma y verificación:

Proceso de firma:

1. A genera una clave k usando la función de hash sobre el mensaje que quiere enviar, es decir $k = h(m)$.
2. A escoge un valor aleatorio v de b bits de longitud, que se usara pare verificar la firma.
3. A escoge una serie de funciones públicas $g_1(), g_2(), ..., g_r()$ de un directorio publico y unos valores aleatorios $x_1, x_2, ..., x_r$ para cada una de estas claves. Para cada función pública genera $y_i = g_i(x_i)$.
4. A continuación empieza desde el principio y desde el final del anillo a realizar las siguientes operaciones hasta llegar a su posición. Desde el principio del anillo genera un valor $t_1 = E_k(y_1 \oplus v)$, a continuación $t_2 = E_k(y_2 \oplus t_1)$, así hasta que llega a su posición en el anillo. Desde el final del anillo genera el valor $t_r = E_k^{-1}(y_r \oplus v)$, a continuación $t_{r-1} = E_k^{-1}(y_{r-1} \oplus t_r)$, así hasta que llega a su posición en el anillo.
5. Una vez llega a su posición obtiene y_s a partir de $y_s = t_{s-1} \oplus t_{s+1}$, y con y_s es capaz de generar $x_s = g_s^{-1}(y_s)$.
6. Por ultimo envía los valores $v, x_1, x_2, ..., x_r$ que contendrá el valor x_s, las funciones públicas $g_1, g_2, ..., g_r$ que contendrá su función pública $g_s(x)$ y el mensaje m.

Proceso de verificación:

1. B recibe los valores enviado por A, produce el valor k con $k = h(m)$.
2. B aplica $E_k(\ g_r(x_r) \oplus ... \ E_k(\ g_2(x_2) \oplus \ E_k(\ g_1(x_1) \oplus \ v)))) = z$
3. Si $z = v$ entonces B da por valida la firma.

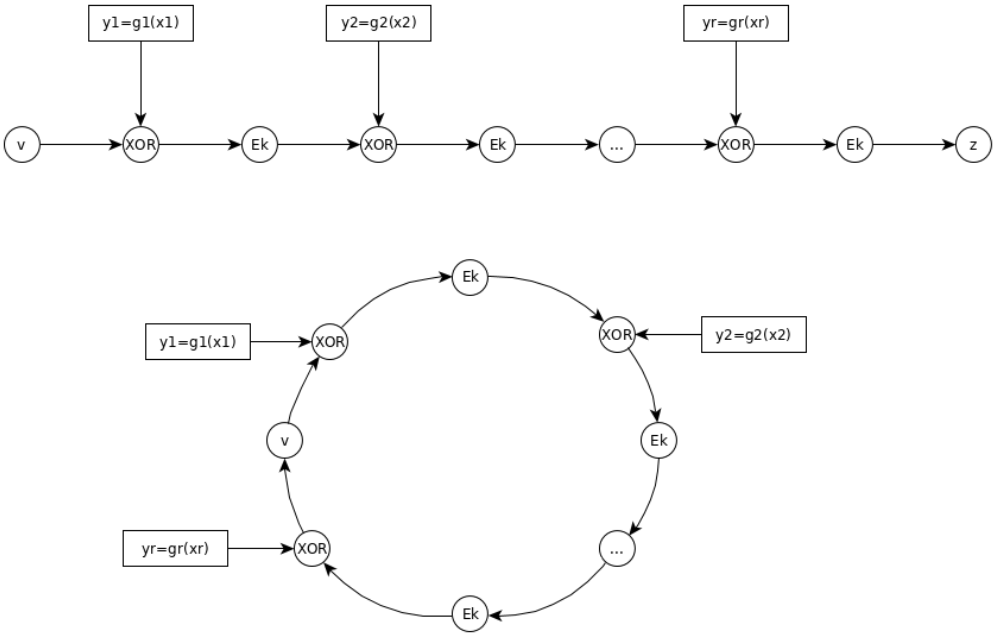


Figura H.1: Funcionamiento de las firmas de anillo.

I

Máquinas de Turing probabilísticas y algoritmos probabilísticos de tiempo polinómico

Una MT probabilística es una máquina de Turing no determinista que en un cierto estado hace una elección, según una distribución de probabilidad, sobre a qué estado tiene transitar. Realmente una MTP se parece a una máquina determinista, pero al usar una distribución de probabilidad para hacer transición de estados se obtienen resultados estocásticos en cada ejecución.

Una MTP usa un algoritmo probabilístico de tiempo polinómico (PPT) para su ejecución.

Los algoritmos probabilísticos de tiempo polinómico son un tipo de algoritmos asociados a una máquina de Turing probabilística que resuelven un problema no determinista de una forma determinista, es decir, son capaces de dar una solución aproximada de un problema no determinista.

El funcionamiento consiste en que en una transición de estado que pueda tener múltiples estados equiprobables se hace una elección con una determinada función de probabilidad sobre a qué estado se va a hacer transición. Por tanto diferentes ejecuciones pueden resultar en distintas soluciones y distintos tiempos de ejecución.

Este tipo de algoritmo son útiles en las pruebas interactivas donde es deseable que se realizan decisiones aleatorias para que no sea posible predecir el comportamiento del algoritmo.

J

Suposiciones sobre problemas criptográficos.

J.1. Grupos bilineales.

Dados dos grupos cíclicos $G_1 = g_1$, $G_2 = g_1$ de orden $Q = k$ siendo Q primo, donde es posible que $G_1 = G_2$ existe una función e, tal que $e : G_1 \times G_2 \Rightarrow G_T$ siendo G_T otro grupo bilineal en Q, G_1 y G_2 serán bilineales si:

- Eficiencia: $e : G_1 \times G_2 \Rightarrow G_T$ puede ser computado en tiempo polinómico.
- Bilinealidad: Para todo $u \in G_1$, $v \in G_2$ y $a, b \in \mathbb{Z}_Q : e(u^a, v^b) = e(u, v)^{a,b}$.
- No degeneración: $e(g_1, g_2) = 1$.

Por tanto puede existir un homomorfismo ψ viable de computar para G_1 yG_2, tal que $\psi(g_2) = g_1$.

J.2. Decisional Diffie-Hellman (DDH)

Decisional Diffie-Hellman, (Boneh, 1998), es una suposición de complejidad computacional sobre logaritmos discretos en grupos cíclicos. Esta suposición está fuertemente relacionada con la complejidad de computar el logaritmo discreto de un número (anexo F) y es la base de varios criptosistemas.

La seguridad de este sistema reside en la apariencia aleatoria que tienen los números pertenecientes a un grupo cíclico. El funcionamiento consiste en que dados un grupo cíclico G con un generador g y unos valores aleatorios a, b y c , tales que, $a, b, c \in [1, |G|]$ intentar distinguir los casos en que la cadena que se transmite está formada por (g^a, g^b, g^c) o por (g^a, g^b, g^{ab}) es muy complejo computacionalemnte, haciendo por tanto al sistema seguro.

J.3. Linear Diffie-Hellman (LDH, DLDH)

Dado un grupo cíclico G_1 y tres generadores aleatorios g_1, g_2 yg_3 de G_1, teniendo la entrada (g_1, g_2 ,g_3, g_1^a, g_2^b ,g_3^c) el problema linear de Diffie-Hellman consiste en averiguar si $a + b = c$ se cumple o no.

Este problema es cumputacionalemnte complejo de resolver por motivos parecidos a los del apartado anterior, de hecho soluciones de este problema son aplicable para DDH.

J.4. *q*-Strong Diffie-Hellman (*q*-SDH)

La suposición q-SDH fue presentada por primera vez en (Boneh et al., 2004) como sistema para generar firmas grupales cortas. En posteriores trabajos se ha usado con la misma idea, como en (Choi et al., 2006).

Su funcionamiento consiste en que dados los grupos cíclicos G_1 y G_2, formados a partir de un número primo de orden p y con sus respectivos generadores g_1 y g_2 y dada la entrada de $q+3$ tuplas:

$$(g_2, g_2^x, g_1, g_1^x, g_1^{x^2}, ..., g_2^{x^q}) \in G_1^{q+1} \times G_2^2 \tag{J.1}$$

obtener la salida $(c, g_1^{\frac{1}{x+c}}) \in \mathbb{Z}_q \times G_1$, tal que $x + c \not\equiv 0$ mód p.

K

Ejemplos de pruebas de conocimiento cero

K.1. Prueba de conocimiento cero para el problema de los tres colores

El problema de los tres colores es un tipo de problema NP completo que consiste en pintar un grafo con solo tres colores y sin que haya dos nodos adyacentes del mismo color. Un ejemplo de una solución valida para este problema se puede ver en la figura K.1

Actualmente encontrar una solución para este problema de una forma determinista implica un tiempo de ejecución exponencial, es decir, si tenemos n nodos el tiempo de ejecución de este problema sería $O(m^n)$, donde m es un valor constante. En realidad es posible que exista una solución de tiempo polinómico, tal que $O(n^m)$, pero no la conocemos, ello implicaría $P = NP$, pero por ahora nadie es capaz de asegurárnoslo, por lo que no vamos a entrar a este tema [1].

La prueba de conocimiento cero para este problema es bastante sencilla. Como siempre

[1] En caso de que en la comisión evaluadora este Xavier Alaman o Jose Dorronsoro espero que estos dos últimos párrafos les sean de su agrado.

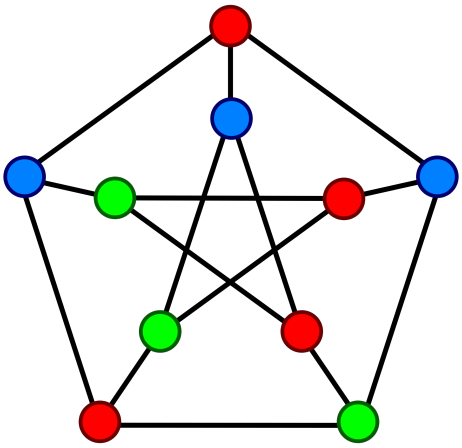


Figura K.1: Un grafo valido para el problema de los 3 colores.

tenemos el rol de verificador V y el de probador P. Como ya se explico en 2.4, P quiere demostrar a V que tiene una solución para este problema de un grafo que el proporciona.

Lo que hará P para demostrar a V que tiene una solución para este problema es tapar el color de todos los nodos y pedirle a V que escoja un eje que una dos nodos para revelarle el color de los dos nodos que lo forman. Si los dos nodos son del mismo color entonces P no tiene una solución para este problema, si por el contrario no son del mismo color es posible que P tenga una solución para este problema. Como ya se ha explicado en 2.4, esta prueba tiene una posibilidad de fallo, por tanto se repite varias veces hasta minimizarla.

Si uno es suficientemente perspicaz se estará dando cuenta de que V es capaz de sacar la solución de P preguntando por distintos ejes hasta obtener todos los nodos. Lo que hace P para evitar esto es no dar los tres colores reales que tienen los nodos, sino sustituirlos por tres colores aleatorios diferentes para cada ejecución de la prueba.

K.2. Prueba de conocimiento cero para q-SDH.

q-SDH es un problema computacional, relacionado con grupos bilineales (Anexo J), complejo de resolver que es usado en varios sistemas de firma grupal como base para una prueba de conocimiento cero.

En (Boneh et al., 2004) se presento uno de los primeros protocolos que hacía uso de q-SDH para una ZKP. De forma resumida, esta prueba consistía en demostrar que se tiene una solución para un problema q-SDH.

El funcionamiento de esta ZKP consiste en que dados los valores públicos $g_1, u, v, h \in G_1$ y $g_2, w \in G_2$ que son generadores para sus respectivos grupos cíclicos, tenemos que $w = g_2^x$ para un valor secreto $x \in \mathbb{Z}_q$. El protocolo prueba que se pose el par (A, c), donde $A \in G_1$ y $c \in \mathbb{Z}_q$, tal que $A^{x+c} = g_1$ y que satisfaga que $e(A, wg_2^c) = e(g_1, g_2)$, es decir que produzcan el mismo generador.

En este protocolo existen un probador P que quiere probar a un verificador V que tiene una solución para un problema q-SDH. P escoge los exponentes $\alpha, \beta \leftarrow \mathbb{Z}_q$ y computa la encriptación lineal A:

$$T_1 \leftarrow u^\alpha \quad T_2 \leftarrow v^\beta \quad T_3 \leftarrow Ah^{\alpha+\beta}$$

y los valores auxiliares $\delta_1 \leftarrow c\alpha$ y $\delta_2 \leftarrow c\beta$.

P y V llevan acabo una ZKP con los valores $(\alpha, \beta, c, \delta_1, \delta_2)$ que cumplen las relaciones:

$$\begin{aligned}
&u^\alpha = T_1 \quad u^\alpha = T_1 \\
&e(T_3, g_2)^c \cdot e(h, w)^{-\alpha-\beta} \cdot e(h, g_2)^{-\delta_1-\delta_2} = e(g_1, g_2)/e(T_3, w) \\
&T_1^c u^{-\delta_1} = 1 \quad T_2^c v^{-\delta_2} = 1
\end{aligned}$$

Para realizar la prueba sin revelar información P ofusca con los valores aleatorios en $r_\alpha, r_\beta, r_c, r_{\delta_1}, , r_{\delta_2} \in \mathbb{Z}_q$ de la siguiente forma:

$$\begin{aligned}
&R_1 \leftarrow u^{r_\alpha} \quad R_2 \leftarrow v^{r_\beta} \\
&R_3 \leftarrow e(T_3, g_2)^{r_c} \cdot e(h, w)^{-r_\alpha - r_\beta} \cdot e(h, g_2)^{-r_{\delta_1} - r_{\delta_2}} \\
&R_4 \leftarrow T_1^{r_c} \cdot u^{-r_{\delta_1}} \quad R_5 \leftarrow T_2^{r_c} \cdot v^{-r_{\delta_2}}
\end{aligned}$$

y manda los valores $(T_1, T_2, T_3, R_1, R_2, R_3, R_4, R_5)$ a V, que le envía un valore de verificación $y \in \mathbb{Z}_q$, P computa y envía de vuelta a V:

$$s_\alpha = r_\alpha + y\alpha$$
$$s_\beta = r_\beta + y\beta$$
$$s_c = r_c + yc$$

V entonces comprueba que se cumpla:

$$u^{s_\alpha} = T_1^y \cdot r_1$$
$$V^{s_\beta} = T_2^y \cdot r_2$$
$$e(T_3, g_2)^{s_c} \cdot e(h, w)^{-s_\alpha - s_\beta} \cdot e(h, g_2)^{-s_{\delta_1} - s_{\delta_2}} = (e(g_1, g_2)/e(T_3, w))^y \cdot R_3$$
$$T_1^{s_c} u^{-s_{\delta_1}} = R_4$$
$$T_2^{s_c} v^{-s_{\delta_2}} = R_5$$

Si las relaciones anteriores se cumplen entonces V aceptara la firma.

L

SSL/TLS

El proceso de handsake de SSL se puede ver en la figura L.1 y consiste en:

1. El cliente envía `client_hello`, que contiene la información de los protocolos que se usaran y el servidor le responde iniciando el proceso de `server_hello`, que contiene su respuesta al uso de estos.

2. El servidor envía, si es necesario, *a*) un certificado de su clave pública `certificate`, *b*) el valor de intercambio de calve asimétrica del servidor `server_key_exchange` y *c*) una petición del certificado del cliente `certificate_request`. Dependiendo de que protocolos concretos se estén usando algunas de las operaciones se diferenciaran entre ellas o no se realizaran. Por último el servidor envía `server_hello_done` dando por terminada su parte del intercambio de los valores de su clave asimétrica.

3. El cliente envía, si es necesario, *a*) un certificado de su clave `certificate`, si ha sido solicitado por el servidor, *b*) el valor de intercambio de la calve asimétrica del cliente y *c*) una verificación del certificado.. En esta ocasión, como en el paso anterior, el cliente solo envía la información que le sea requerida.

4. Por último el cliente envía los mensajes `change_cipher_spec` y `finished`, a los que el servidor responde con `change_cipher_spec` y `finished`. En este punto el proceso de handsake habrá terminado.

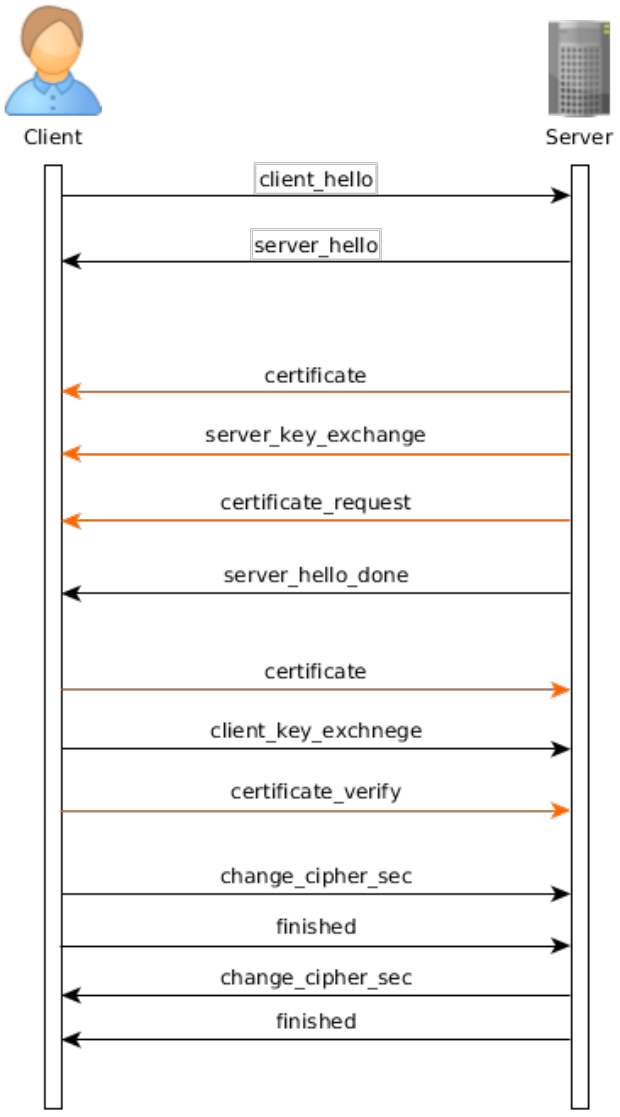


Figura L.1: handsake SSL.

M

Aspecto del programa

Los programas desarrollados para probar el protocolo consisten en un servidor de chat y un cliente que se conecte a ese servidor de chat.

El servidor es un programa sencillo que no admite argumentos de entrada. Requiere que las claves RSA hayan sido generadas y que el grupo también este generado. Para ejecutarlo solo hay que usar el comando:

```
server
```

Por otro lado el cliente es un programa con una interfaz para el usuario, que requiere de la clave pública del servidor, la clave de grupo y la clave de miembro. El programa se ejecuta con:

```
cliente -ip arg -port arg -nick arg -memkey arg
```

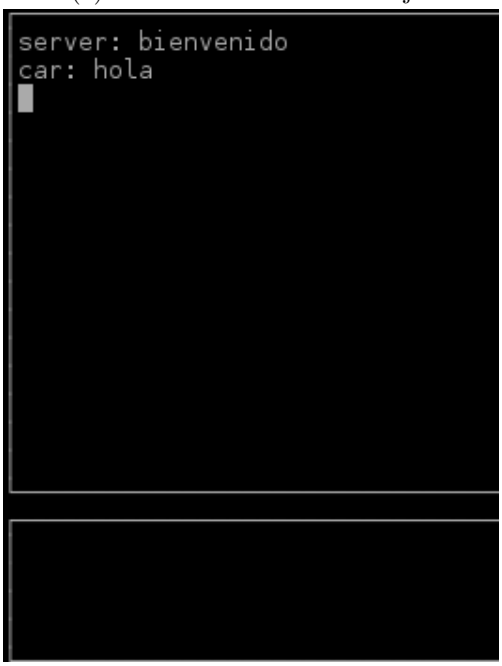
Donde se requiere la ip del servidor (**ip**) donde conectarse, el puerto (**port**), el seudónimo que usara el cliente (**nick**) y el archivo con la clave de miembro (**memkey**) que usara el cliente. Una vez iniciada la conexión satisfactoriamente el cliente se le presentara una interfaz donde podrá ver la gente que se conecta (figura M.1), enviar mensajes (figura M.2) escribiéndolos en la interfaz, cambiarse de nombre(figura M.3) con el comando `/NICK`, y desconectarse mediante el comando `/DISCONNECT` y ser informado de las desconexiones de otros usuarios (figura M.4) .

(a) Cliente recibe mensaje de conexión.

(b) Resultado

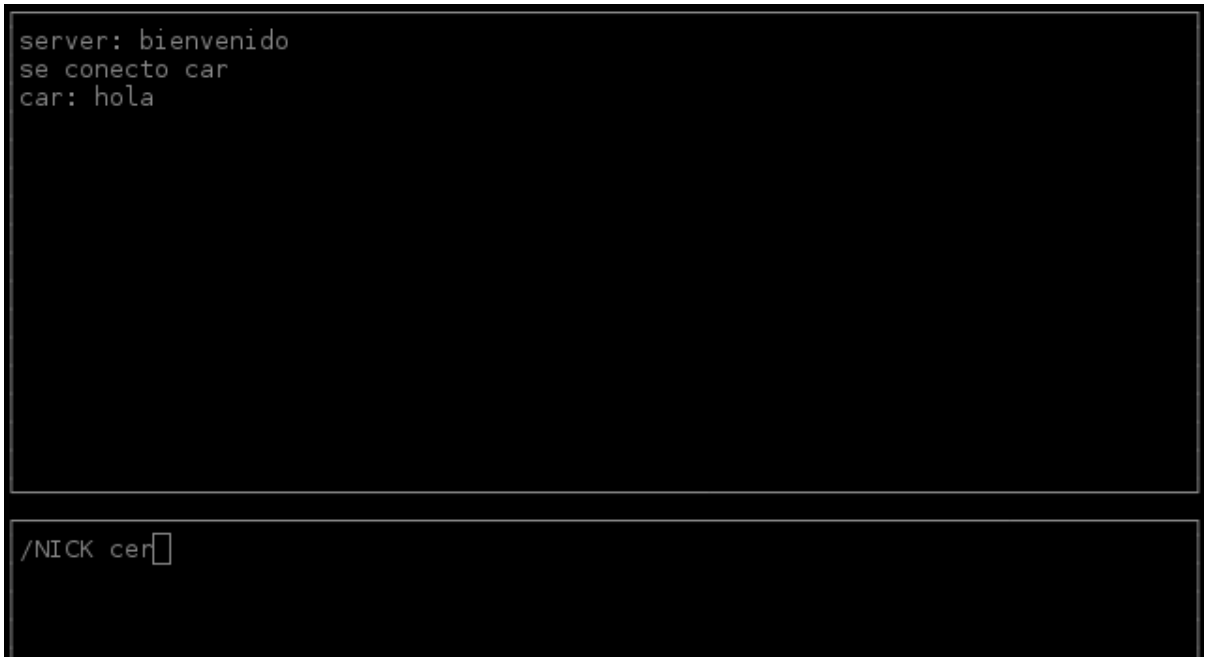
Figura M.1: Cliente se conecta.

(a) Cliente escribe un mensaje.



(b) Cliente recibe un mensaje.

Figura M.2: Envio de mensajes.



(a) Comando.

(b) Resultado

Figura M.3: Funcionamiento del cambio de nombre.

Figura M.4: Desconexión.

N

Codigos

Todos los códigos de este programa se encuentran en el repositorio publico `https://github.com/made1993/chatlibgroupsig` para ser descargados libremente. En caso de duda ponerse en contacto con mariogr_93@hotmail.com

N.1. Código de AES_CBC_256

```
#include "../include/funcionesAES.h"

EVP_CIPHER_CTX* create_ctx(){
  EVP_CIPHER_CTX* ctx = NULL;
  ctx = EVP_CIPHER_CTX_new();
  EVP_CIPHER_CTX_init(ctx);

  return ctx;
}

int encrypt_cbc256(EVP_CIPHER_CTX* ctx, unsigned char* key, unsigned char* iv,
      const unsigned char* text, unsigned char** out, int textlen){

  int tmp = 0, ciphlen = 0;

  if(ctx == NULL || key == NULL || iv == NULL ||
    text == NULL || out == NULL || textlen<1)
    return 0;

  EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
  *out = calloc(textlen+64, sizeof(char));

  EVP_EncryptUpdate(ctx, *out, &tmp, text, textlen);
  ciphlen = tmp;

  EVP_EncryptFinal_ex(ctx, (*out) + tmp, &tmp);
  ciphlen += tmp;
  EVP_CIPHER_CTX_cleanup(ctx);
  *out = realloc(*out, ciphlen);
  return ciphlen;
}
```

```
33 int decrypt_cbc256(EVP_CIPHER_CTX* ctx, unsigned char* key, unsigned char* iv,
       const unsigned char* text, unsigned char** out, int textlen){
35
     int tmp = 0 , deciphlen = 0;
37
     if(ctx == NULL || key == NULL || iv == NULL ||
39     text == NULL || out == NULL || textlen<1)
       return 0;
41
     EVP_DecryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv);
43   *out = calloc(textlen, sizeof(char));
45   EVP_DecryptUpdate(ctx, *out, &tmp, text, textlen);
     deciphlen = tmp;
47
     EVP_DecryptFinal_ex(ctx, (*out) +tmp, &tmp);
49   deciphlen += tmp;
51   *out = realloc(*out, deciphlen);
     EVP_CIPHER_CTX_cleanup(ctx);
53
     return deciphlen;
55
}
```

codigos/funcionesAES.c

N.2. Código de intercambio de Diffie-Hellman.

```
2 #include "../include/funcionesDH.h"
4
6 int getParamsIniDH(EVP_PKEY** params){
    if(params == NULL)
8     return 0;
10   *params = NULL;
     if(NULL == (*params = EVP_PKEY_new()))
12     return 0;
14   if(1 != EVP_PKEY_set1_DH(*params,DH_get_2048_256()))
       return 0;
16
     return 1;
18 }
20 int genNewParamsIniDH(EVP_PKEY** params, EVP_PKEY_CTX** pctx){
22   if(pctx == NULL || params == NULL)
       return 0;
24   *pctx = EVP_PKEY_CTX_new_id(EVP_PKEY_DH, NULL);
     if(!*pctx){
26     return 0;
     }
28
     if(1 != EVP_PKEY_paramgen_init(*pctx)){
30     EVP_PKEY_CTX_free(*pctx);
       return 0;
```

```
32    }
34    if(1 != EVP_PKEY_CTX_set_dh_paramgen_prime_len(*pctx, 2048)){
        EVP_PKEY_CTX_free(*pctx);
36      return 0;
      }
38
      if(1 != EVP_PKEY_paramgen(*pctx, params)){
40      EVP_PKEY_CTX_free(*pctx);
        return 0;
42    }
44    return 1;
  }
46
  int genKeyFromParamsDH(EVP_PKEY_CTX** kctx, EVP_PKEY** dhkey, EVP_PKEY* params){
48
      if(kctx == NULL || dhkey == NULL || params == NULL)
50
    *dhkey = NULL;
52  if(NULL == (*dhkey = EVP_PKEY_new()))
      return 0;
54
    /* Create context*/
56  if(!(*kctx = EVP_PKEY_CTX_new(params, NULL))){
      return 0;
58  }
60  /* Generate a new key */
    if(1 != EVP_PKEY_keygen_init(*kctx)){
62    EVP_PKEY_CTX_free(*kctx);
      return 0;
64  }
66  if(1 != EVP_PKEY_keygen(*kctx, dhkey)){
      EVP_PKEY_CTX_free(*kctx);
68    return 0;
    }
70
    return 1;
72  }
  unsigned  char* deriveSharedSecretDH(EVP_PKEY* privkey, EVP_PKEY* peerkey){
74    unsigned char* skey = NULL;
    EVP_PKEY_CTX* ctx = NULL;
76  size_t skeylen;
    if(privkey == NULL || peerkey == NULL)
78    return 0;
80  ctx = EVP_PKEY_CTX_new(privkey, NULL);
    if (!ctx){
82    return NULL;
    }
84  if(EVP_PKEY_derive_init(ctx) <= 0){
      EVP_PKEY_CTX_free(ctx);
86    return NULL;
    }
88
    if (EVP_PKEY_derive_set_peer(ctx, peerkey) <= 0){
90    EVP_PKEY_CTX_free(ctx);
      return NULL;
92  }
94  /* Determine buffer length */
```

```
  if (EVP_PKEY_derive(ctx, NULL, &skeylen) <= 0){
96    EVP_PKEY_CTX_free(ctx);
      return NULL;
98  }

100 skey = OPENSSL_malloc(skeylen);

102 if (!skey){
      EVP_PKEY_CTX_free(ctx);
104   return NULL;
    }
106
    if (EVP_PKEY_derive(ctx, skey, &skeylen) <= 0){
108   OPENSSL_free(skey);
      EVP_PKEY_CTX_free(ctx);
110   return NULL;
    }
112
    return skey;
114 }

116 int DHpubKeyToMsg(EVP_PKEY* pubKey, char ** msg){
    if(pubKey == NULL || msg == NULL)
118   return 0;
    *msg = NULL;
120 return i2d_PUBKEY(pubKey, (unsigned char**) msg);
    }
122


124

    int msgToDHpubKey(EVP_PKEY** pubKey, char * msg, int msglen){
126 if(pubKey == NULL || msg == NULL || msglen < 1)
      return 0;
128
    *pubKey = EVP_PKEY_new();
130 if(*pubKey == NULL){
      printf("ERR EVP_PKEY_new\n");
132   return 0;
    }
134 d2i_PUBKEY(pubKey, (const unsigned char**) &msg, msglen);
    return 1;
136 }
```

codigos/funcionesDH.c

N.3. Código para el uso de RSA.

```
#include "../include/funcionesRSA.h"
2



4

const char hn[] = "SHA256";
6

int generateKeysRSA(EVP_PKEY** privKey, EVP_PKEY** pubKey){
8   RSA* rsa = NULL;
    if(privKey == NULL || pubKey == NULL)
10    return 0;

12  *privKey = EVP_PKEY_new();
    if(*privKey == NULL){
14    printf("ERR EVP_PKEY_new\n");
```

```
    return 0;
  }

  *pubKey = EVP_PKEY_new();
  if(*pubKey == NULL){
    printf("ERR EVP_PKEY_new\n");
    return 0;
  }

  rsa = RSA_generate_key(2048, RSA_F4, NULL, NULL);

  if(rsa == NULL){
    printf("ERR RSA_generate_key\n");
    return 0;
  }

  if(1 != EVP_PKEY_assign_RSA(*privKey,
            RSAPrivateKey_dup(rsa))){

    printf("ERR EVP_PKEY_assign_RSA\n");
    return 0;
  }

  if(1 != EVP_PKEY_assign_RSA(*pubKey,
            RSAPublicKey_dup(rsa))){

    printf("ERR EVP_PKEY_assign_RSA\n");
    return 0;
  }
  return 1;
}


int signMsgRSA(EVP_PKEY* key, const unsigned char* msg,
  unsigned char** sig, size_t* slen, size_t msglen){

  EVP_MD_CTX* ctx = NULL;
  const EVP_MD* md = NULL;

  if(msg == NULL || sig == NULL || slen == NULL)
    return 0;

  ctx = EVP_MD_CTX_create();
  md = EVP_get_digestbyname(hn);

  if(md == NULL){
    printf("ERR EVP_get_digestbyname\n");
    return 0;
  }
  if(ctx == NULL){
    printf("ERR EVP_MD_CTX_create\n");
    return 0;
  }

  if(1 != EVP_DigestInit_ex(ctx, md, NULL)){
    printf("ERR EVP_DigestInit_ex\n");
    return 0;
  }
  if(1 != EVP_DigestSignInit(ctx, NULL, md, NULL, key)){
    printf("ERR EVP_DigestSignInit\n");
    return 0;
  }
```

```
78    /*SE FIRMA EL MENSAJE*/
      if(1 != EVP_DigestSignUpdate(ctx, msg, msglen)){
80      printf("ERR EVP_DigestSignUpdate\n");
        return 0;
82    }

84    if(1 != EVP_DigestSignFinal(ctx, NULL, slen)){
        printf("ERR EVP_DigestSignFinal\n");
86      return 0;
      }
88    *sig = OPENSSL_malloc(*slen);

90    if(*sig == NULL){
        printf("ERR OPENSSL_malloc\n");
92    }

94    if(1 != EVP_DigestSignFinal(ctx, *sig, slen)){
        printf("ERR EVP_DigestSignFinal\n");
96      return 0;
      }
98
      EVP_MD_CTX_destroy(ctx);
100   ctx = NULL;

102   return 1;

104 }
    int verifySignRSA(EVP_PKEY* key, const unsigned char* sig,
106   const unsigned char* msg, size_t slen, size_t msglen){

108   EVP_MD_CTX* ctx = NULL;
      const EVP_MD* md = NULL;
110
      if(!msg || !sig || !slen || !key) {
112         return -1;
        }
114
      ctx = EVP_MD_CTX_create();
116   md = EVP_get_digestbyname(hn);
      if(md == NULL){
118     printf("ERR EVP_get_digestbyname\n");
        return 0;
120   }
      if(ctx == NULL){
122     printf("ERR EVP_MD_CTX_create\n");
        return 0;
124   }
      if(1 != EVP_DigestInit_ex(ctx, md, NULL)){
126     printf("ERR EVP_DigestInit_ex\n");
        return 0;
128   }
      if(1 != EVP_DigestVerifyInit(ctx, NULL, md, NULL, key)){
130     printf("ERR EVP_DigestVerifyInit\n");
        return 0;
132   }

134   if(1 != EVP_DigestVerifyUpdate(ctx, msg, msglen)){
        printf("ERR EVP_DigestVerifyUpdate\n");
136     return 0;
      }
138   ERR_clear_error();

140
```

```
  return EVP_DigestVerifyFinal(ctx, sig, slen);
}

int reciveRSAsign(int sockfd, EVP_PKEY* pubKey,
  unsigned char** msg){

  char* buff;
  unsigned char* sig = NULL,* auxMsg;
  int msglen = 0;
  if(sockfd <1 || pubKey == NULL || msg == NULL)
    return 0;
  msglen = recibir(sockfd, &buff);
  if(msglen == -1)
    return 0;
  sig = malloc(sizeof(char) * SHA256_SIGLEN);
  auxMsg =  malloc(sizeof(char) * (msglen - SHA256_SIGLEN));

  if(auxMsg == NULL || sig == NULL){
    return 0;
  }

  memcpy(sig, buff, sizeof(char) * SHA256_SIGLEN);
  memcpy(auxMsg, &buff[SHA256_SIGLEN], msglen - SHA256_SIGLEN);

  *msg = auxMsg;
  if( !verifySignRSA(pubKey, (const unsigned char*)sig,
  (const unsigned char*) auxMsg,  SHA256_SIGLEN, msglen - SHA256_SIGLEN))

    return 0;
  return msglen - SHA256_SIGLEN;
}


int sendRSAsign(int sockfd, EVP_PKEY* privKey,
  const unsigned char* msg, int msglen){

  unsigned char* buff = NULL;
  size_t slen = 0;
  /*CONTROL DE ERRORES*/
  if(privKey == NULL || msg == NULL || msglen <= 0){
    return 0;
  }

  /*SE FIRMA EL MENSAJE*/
  if (1 != signMsgRSA(privKey, msg, &buff, &slen, msglen)){
    return 0;
  }

  buff = realloc(buff, sizeof(char) * (msglen + slen));

  memcpy(&buff[slen], msg, msglen);
  if(-1 == escribir(sockfd, (char*)buff, slen + msglen)){
    return 0;
  }

  return 1;
}


int msgToRSApubKey(EVP_PKEY** pubKey, char* msg, int msglen){
  if(pubKey == NULL || msg == NULL || msglen < 1)
    return 0;
```

```
204   *pubKey = EVP_PKEY_new();
      if(*pubKey == NULL){
206     printf("ERR EVP_PKEY_new\n");
        return 0;
208   }
      d2i_PUBKEY(pubKey, (const unsigned char**) &msg, msglen);
210   return 1;

212 }

214 int RSApubKeyToMsg(EVP_PKEY* pubKey, char** msg, int* msglen){

216   if(pubKey == NULL || msglen == NULL || msg == NULL)
        return 0;
218
      *msglen = i2d_PUBKEY(pubKey, (unsigned char **)msg);
220
      return 1;
222 }

224 int RSAfileToPubKey(EVP_PKEY** pubKey, char* fname){
      FILE * f = NULL;
226   if(pubKey == NULL || fname == NULL || strlen(fname)<1)
        return 0;
228   f = fopen(fname, "r");
      if (f == NULL)
230     return 0;
      *pubKey = EVP_PKEY_new();
232   if(*pubKey == NULL){
        printf("ERR EVP_PKEY_new\n");
234     return 0;
      }
236   PEM_read_PUBKEY(f, pubKey,  NULL, NULL);
      fclose(f);
238   if(*pubKey == NULL)
        return 0;
240   return 1;
}
242
int RSApubKeyToFile(EVP_PKEY* pubKey, char* fname, int* msglen){
244   FILE * f = NULL;
      if(pubKey == NULL || fname == NULL ||
246     strlen(fname)<1 || msglen == NULL)

248       return 0;
      f = fopen(fname, "w");
250   if (f == NULL)
        return 0;
252   *msglen = PEM_write_PUBKEY(f, pubKey);
      //*msglen = i2d_PUBKEY_fp(f, pubKey);
254   fclose(f);
      if(*msglen < 1)
256     return 0;
      return 1;
258
}
260
int RSAfileToPrivKey(EVP_PKEY** privKey, char* fname){
262   FILE * f = NULL;
      if(privKey == NULL || fname == NULL || strlen(fname)<1)
264       return 0;
      f = fopen(fname, "r");
266   if (f == NULL)
```

```
    return 0;
  *privKey = EVP_PKEY_new();
  if(*privKey == NULL){
    printf("ERR EVP_PKEY_new\n");
    return 0;
  }
  PEM_read_PrivateKey(f, privKey, NULL, NULL);
  fclose(f);
  if(*privKey == NULL)
    return 0;
  return 1;
}

int RSAprivKeyToFile(EVP_PKEY* privKey, char* fname, int* msglen){
  FILE * f = NULL;
  if(privKey == NULL || fname == NULL ||
    strlen(fname)<1 || msglen == NULL)

     return 0;
  f = fopen(fname, "w");
  if (f == NULL)
    return 0;
  *msglen =  PEM_write_PrivateKey(f , privKey, NULL, NULL, 0, 0, NULL);
  fclose(f);
  if(*msglen < 1)
    return 0;
  return 1;

}

int reciveRSAkey(int sockfd, EVP_PKEY** pubKey){
  char* buffKey;
  int keylen = 0;

  if(sockfd <= 0 || pubKey== NULL || *pubKey == NULL)
    return 0;

  keylen = recibir(sockfd, &buffKey);
  if (keylen == -1)
    return 0;

  msgToRSApubKey(pubKey, buffKey, keylen);
  return 1;
}

int sendRSAkey(int sockfd, EVP_PKEY* pubKey){
  char* buffKey = NULL;
  int bufflen = 0;

  if(!RSApubKeyToMsg(pubKey, &buffKey, &bufflen)) return 0;

  escribir(sockfd, buffKey, bufflen);
  free(buffKey);
  return 1;
}
```

codigos/funcionesRSA.c

N.4. Código para el uso de firmas grupales.

```
#include "../include/funcionesGS.h"


int import_manager(groupsig_key_t** grpkey, groupsig_key_t** mgrkey, crl_t** crl,
     gml_t** gml,
       char* grpkeyf, char* mgrkeyf, char* crlf, char* gmlf, int scheme){

  int key_format = -1;

  if(grpkeyf == NULL || mgrkeyf == NULL || gmlf == NULL || crlf == NULL ||
    strlen(grpkeyf) < 1 || strlen(mgrkeyf) < 1 || strlen(crlf) < 1 || strlen(gmlf
    ) < 1){
    return 0;
  }
  switch(scheme) {
  case GROUPSIG_KTY04_CODE:
    key_format = GROUPSIG_KEY_FORMAT_FILE_NULL_B64;
    break;
  case GROUPSIG_BBS04_CODE:

  case GROUPSIG_CPY06_CODE:

    key_format = GROUPSIG_KEY_FORMAT_FILE_NULL;
    break;
  default:
    fprintf(stderr, "Error: unknown scheme.\n");
    return 0;
  }

  *grpkey = NULL; *mgrkey = NULL; *crl = NULL; *gml =  NULL;

  *grpkey = groupsig_grp_key_import(scheme, key_format, grpkeyf);
  if(*grpkey == NULL){
    fprintf(stderr, "Error: invalid grpkey %s.\n", grpkeyf);
    return 0;
  }

  *mgrkey = groupsig_mgr_key_import(scheme, key_format, mgrkeyf);
  if(*mgrkey == NULL){
    fprintf(stderr, "Error: invalid mgrkey %s.\n", mgrkeyf);
    return 0;
  }

  *crl = crl_import(scheme, CRL_FILE, crlf);
  if(*crl == NULL){
    fprintf(stderr, "Error: invalid crl %s.\n", crlf);
    return 0;
  }

  *gml = gml_import(scheme, GML_FILE, gmlf);
  if(*gml == NULL){
    fprintf(stderr, "Error: invalid gml %s.\n", gmlf);
    return 0;
  }

  return 1;

}

int import_member(groupsig_key_t** grpkey, groupsig_key_t** memkey, char* grpkeyf
    , char* memkeyf, int scheme){
```

```
60    int key_format = -1;

62    if(grpkeyf == NULL || memkeyf == NULL || strlen(grpkeyf) < 1 || strlen(memkeyf)
        < 1 ){
        return 0;
64    }
      switch(scheme) {
66    case GROUPSIG_KTY04_CODE:
        key_format = GROUPSIG_KEY_FORMAT_FILE_NULL_B64;
68      break;
      case GROUPSIG_BBS04_CODE:
70
      case GROUPSIG_CPY06_CODE:
72
        key_format = GROUPSIG_KEY_FORMAT_FILE_NULL;
74      break;
      default:
76      fprintf(stderr, "Error: unknown scheme.\n");
        return 0;
78    }

80    *grpkey = NULL; *memkey = NULL;

82    *grpkey = groupsig_grp_key_import(scheme, key_format, grpkeyf);
      if(*grpkey == NULL){
84      fprintf(stderr, "Error: invalid grpkey %s.\n", grpkeyf);
        return 0;
86    }
      *memkey = groupsig_mem_key_import(scheme, key_format, memkeyf);
88    if(*memkey == NULL){
        fprintf(stderr, "Error: invalid memkey %s.\n", memkeyf);
90      return 0;
      }
92
      return 1;
94  }


96
  int signMsgGS(groupsig_key_t* grpkey, groupsig_key_t* memkey, uint8_t scheme,
98          char *msgstr, char** sigstr){
      message_t* msg = NULL, *sigmsg =  NULL;
100   groupsig_signature_t* sig = NULL;
      int size;
102   if(grpkey == NULL || memkey == NULL || msgstr == NULL || sigstr == NULL){
        fprintf(stderr, "Error: failed to sign a message.\n");
104     return IERROR;
      }
106   *sigstr = NULL;
      msg = message_from_string(msgstr);
108   if(msg == NULL){
        fprintf(stderr, "Error: failed to initialize the message object.\n");
110     return IERROR;
      }
112
      if(!(sig = groupsig_signature_init(scheme))) {
114     fprintf(stderr, "Error: failed to initialize the group signature object.\n");
        return IERROR;
116   }

118   if(groupsig_sign(sig, msg, memkey, grpkey, UINT_MAX) == IERROR){
        fprintf(stderr, "Error: signing failure.\n");
120     return IERROR;
      }
```

```
122
124
  sigmsg = message_init();
126  groupsig_signature_export(sig, GROUPSIG_SIGNATURE_FORMAT_MESSAGE_NULL, sigmsg);
  groupsig_signature_free(sig);
128
  *sigstr = malloc(sigmsg->length + sizeof(uint64_t));
130  memcpy(*sigstr, &sigmsg->length, sizeof(uint64_t));
  memcpy((*sigstr) + sizeof(uint64_t), sigmsg->bytes, sigmsg->length);
132  /*TODO quiza haya que meter el esquema que se usa*/

134  size = (int)sigmsg->length + sizeof(uint64_t);
  message_free(msg);
136  message_free(sigmsg);
  return size;
138 }

140 int verifySignGS(char* sigstr, groupsig_key_t *grpkey, char* msgstr, uint8_t
    scheme){

142  uint8_t bool = 0;
  groupsig_signature_t* sig = NULL;
144  message_t *msg = NULL, *sigmsg = NULL;

146  if(sigstr == NULL || grpkey == NULL || msgstr == NULL){
    fprintf(stderr, "Error: NULL input.\n");
148    return IERROR;
  }
150
  sigmsg =  message_init();
152  memcpy(&sigmsg->length, sigstr, sizeof(uint64_t));
  sigmsg->bytes = malloc(sigmsg->length);
154  memcpy(sigmsg->bytes, sigstr + sizeof(uint64_t), sigmsg->length);

156  if((sig = groupsig_signature_import(scheme,
     GROUPSIG_SIGNATURE_FORMAT_MESSAGE_NULL, sigmsg)) ==  NULL){
     printf("Error: failed to import the signature.\n" );
158    return IERROR;
  }
160
  msg = message_from_string(msgstr);
162  if(msg == NULL){
    fprintf(stderr, "Error: failed to initialize the message object.\n");
164    return IERROR;
  }
166  if(groupsig_verify(&bool, sig, msg, grpkey) == IERROR) {
    fprintf(stderr, "Error: verification failure.\n");
168    return IERROR;
  }
170  message_free(msg);
  message_free(sigmsg);
172  groupsig_signature_free(sig);
  return bool;
174 }

176
int traceSignGS(char* sigstr, groupsig_key_t *grpkey, groupsig_key_t *mgrkey,
    crl_t* crl,
178      gml_t* gml, uint8_t scheme){
  uint8_t bool = 0;
180  message_t * sigmsg = NULL;
  groupsig_signature_t* sig = NULL;
```

```
182
    if(sigstr == NULL || grpkey == NULL || mgrkey == NULL ||
184   crl == NULL || gml == NULL){
    fprintf(stderr, "Error: NULL input.\n");
186   return IERROR;
  }
188
  sigmsg =  message_init();
190 memcpy(&sigmsg->length, sigstr, sizeof(uint64_t));
  sigmsg->bytes = malloc(sigmsg->length);
192 memcpy(sigmsg->bytes, sigstr + sizeof(uint64_t), sigmsg->length);

194 if((sig = groupsig_signature_import(scheme,
    GROUPSIG_SIGNATURE_FORMAT_MESSAGE_NULL, sigmsg)) ==  NULL){
    printf("Error: failed to import the signature.\n" );
196   return IERROR;
  }
198
  if(groupsig_trace(&bool, sig, grpkey, crl, mgrkey, gml) == IERROR) {
200   fprintf(stderr, "Error: failed to trace the signature.\n");
    return 1;
202   }
    message_free(sigmsg);
204 groupsig_signature_free(sig);
    return bool;
206 }


208
int revokeSigGS(groupsig_signature_t *sig, groupsig_key_t *grpkey, groupsig_key_t
    *mgrkey,
210       gml_t *gml,  crl_t *crl, uint8_t scheme, char *s_crl){
  identity_t *id = NULL;
212 int rc = 0;
  trapdoor_t *trap = NULL;
214 if (sig == NULL || grpkey == NULL || mgrkey == NULL || gml == NULL || crl ==
    NULL)
    return IERROR;
216
  if(!(id = identity_init(scheme))){
218   fprintf(stderr, "Error creating identity.\n");
    return IERROR;
220 }
  if((rc = groupsig_open(id, NULL, NULL, sig, grpkey, mgrkey, gml)) == IERROR) {
222   fprintf(stderr, "Error opening signature.\n");
    return IERROR;
224   }
  if(!(trap = trapdoor_init(scheme))) {
226   fprintf(stderr, "Error creating trapdoor.\n");
    return IERROR;
228 }

230 if(groupsig_reveal(trap, crl, gml, *(uint64_t *) id->id) == IERROR) {
    fprintf(stderr, "Error in reveal.\n");
232   return IERROR;
  }
234
  if(crl_export(crl, s_crl, CRL_FILE) == IERROR) {
236   fprintf(stderr, "Error exporting CRL.\n");
    return IERROR;
238 }
  identity_free(id); id = NULL;
240 trapdoor_free(trap); trap = NULL;
```

```
242   return IOK;
}
244
int sigMsgToStrGS(char * msgstr, int msglen, char* sigstr, int siglen, char** dst
    ){
246   int size = -1;
   if(msgstr == NULL || msglen < 1 || sigstr == NULL || siglen < 1 || dst == NULL)
     {
248     fprintf(stderr, "Error: NULL input.\n");
     return IERROR;
250   }
   *dst = NULL;
252   size = msglen + siglen + sizeof(int)*2;
   *dst = malloc (size);
254


256   memcpy(*dst, &siglen, sizeof(int));
   memcpy((*dst) + sizeof(int), sigstr, siglen);
258
   memcpy((*dst) + sizeof(int) + siglen, &msglen, sizeof(int));
260   memcpy((*dst) + sizeof(int)*2 + siglen, msgstr, msglen);

262   return size;
}
264
int strToSigMsgGS(char** msgstr, int *msglen, char** sigstr, int* siglen, char*
    src, int srclen){
266
   if(msgstr == NULL || msglen == NULL || sigstr == NULL || siglen == NULL || src
     == NULL){
268     fprintf(stdout, "Error: NULL input.\n");
     return IERROR;
270   }
   memcpy(siglen, src, sizeof(int));
272   if(*siglen > srclen - sizeof(int)){
     fprintf(stdout, "Error: ivalid size of signature %d\n", *siglen);
274     return IERROR;
   }
276   *sigstr = malloc(*siglen);
   memcpy(*sigstr, src + sizeof(int), *siglen);
278
   memcpy(msglen, src + sizeof(int) + *siglen, sizeof(int));
280   if(*msglen > srclen - *siglen - sizeof(int)*2){
     fprintf(stdout, "Error: ivalid size of message %d\n", *msglen);
282     return IERROR;
   }
284   *msgstr = malloc(*msglen);
   memcpy(*msgstr, src + sizeof(int)*2 + *siglen, *msglen);
286
   return IOK;
288 }
```

codigos/funcionesGS.c

N.5. Código para la conexión

```
1 #ifndef SCONEXION_H
  #define SCONEXION_H
3 #include "conexion.h"
  #include "funcionesAES.h"
5 #include "funcionesRSA.h"
```

```
#include "funcionesGS.h"
#include "funcionesDH.h"

typedef struct SCONEXION
{
  int socket;
  EVP_CIPHER_CTX* ctx;
  unsigned char* key;
  unsigned char* iv;
  groupsig_key_t *grpkey;
  groupsig_key_t *memkey;
  int scheme;
  EVP_PKEY* keyRSA;
  char* lastsig;
  int siglen;
}Sconexion_t;

Sconexion_t* initSconexion(int socket, groupsig_key_t *grpkey,
  groupsig_key_t *memkey, int scheme, EVP_PKEY* keyRSA);

int freeSconexion(Sconexion_t* scnx);

int sendClientCiphMsg(Sconexion_t* scnx,
  const unsigned char* text, int textlen);

int reciveServerCiphMsg(Sconexion_t* scnx, char** msg);


int sendServerCiphMsg(Sconexion_t* scnx,
  const unsigned char* text, int textlen);

int reciveClientCiphMsg(Sconexion_t* scnx, char** msg);

int revokeClient(Sconexion_t* scnx, groupsig_key_t *mgrkey, gml_t *gml, crl_t *
    crl, char* s_crlf);

int clientInitSConexion(Sconexion_t* scnx);


int serverInitSConexion(Sconexion_t* scnx, groupsig_key_t *mgrkey,
  crl_t* crl, gml_t* gml);

int crlRelaod(Sconexion_t* scnx, crl_t** crl, char* s_crl);

#endif
```

codigos/sconexion.h

```
#include "../include/sconexion.h"

#ifdef TIMETESTD

#endif
Sconexion_t* initSconexion(int socket, groupsig_key_t *grpkey,
    groupsig_key_t *memkey, int scheme, EVP_PKEY* keyRSA){
  Sconexion_t* scnx;
  if(socket < 0 || grpkey == NULL  || keyRSA == NULL)
    return NULL;
  scnx =  malloc(sizeof(Sconexion_t));
  scnx->socket = socket;
  scnx->grpkey = grpkey;
  scnx->memkey = memkey;
  scnx->scheme = scheme;
```

```
  scnx->keyRSA = keyRSA;
  scnx->key = NULL;
  scnx->iv = NULL;
  scnx->ctx = create_ctx();
  scnx->lastsig = NULL;
  scnx->siglen = 0;
  return scnx;
}

int freeSconexion(Sconexion_t* scnx){
  if(scnx == NULL)
    return 1;
  close(scnx->socket);
  if(scnx->ctx != NULL){
    free(scnx->ctx); scnx->ctx = NULL;
  }

  if(scnx->key != NULL){
    free(scnx->key); scnx->key = NULL;
  }

  if(scnx->iv != NULL){
    free(scnx->iv); scnx->iv = NULL;
  }

  if(scnx->memkey != NULL){
    free(scnx->memkey); scnx->memkey = NULL;
  }
  return 1;
}


int sendClientCiphMsg(Sconexion_t* scnx,const unsigned char* text, int textlen){
  int msglen = 0, bufflen = 0, siglen = 0;
  char*  msg = NULL, *buff = NULL, *sigstr;
  //FILE* f = NULL;

  #ifdef TIMETESTD
  clock_t ini, fin, totc, tots;
  FILE * f = NULL;
  f =fopen("tmedioCd.dat","a");
  #endif

  if(scnx == NULL || scnx->socket == -1 || scnx->ctx == NULL || scnx->key == NULL
     ||
     scnx->iv == NULL || text == NULL || textlen < 1 || scnx->grpkey == NULL ||
     scnx->memkey == NULL)
     return 0;
  //f = fopen("csend.txt", "a");

  #ifdef TIMETESTD
  ini =  clock();
  siglen = signMsgGS(scnx->grpkey, scnx->memkey, scnx->scheme, (char*)text, &
    sigstr);
  msglen = sigMsgToStrGS((char*)text, textlen, sigstr, siglen, &msg);
  fin =  clock();
  tots = fin - ini;
  #else
  siglen = signMsgGS(scnx->grpkey, scnx->memkey, scnx->scheme, (char*)text, &
    sigstr);
  msglen = sigMsgToStrGS((char*)text, textlen, sigstr, siglen, &msg);
  #endif

```

```
  #ifdef TIMETESTD
77  ini =  clock();
    bufflen = encrypt_cbc256(scnx->ctx, scnx->key, scnx->iv, (const unsigned char*)
      msg,
79          (unsigned char**)&buff, msglen);

81  fin =  clock();
    totc = fin - ini;
83  fprintf(f, "%d %d\n", tots, totc);
    fclose(f);
85  #else

87  bufflen = encrypt_cbc256(scnx->ctx, scnx->key, scnx->iv, (const unsigned char*)
      msg,
          (unsigned char**)&buff, msglen);
89  #endif


91


93  //fprintf(f, "bufflen: %d siglen:%d msg:%d :%s\n", bufflen, siglen , textlen,
      text);
    //fclose(f);
95


97  if(bufflen <1)
      return 0;
99  msglen = escribir(scnx->socket, buff, bufflen);
    scnx->iv = realloc(scnx->iv, bufflen);
101 memcpy(scnx->iv, buff, bufflen);

103 free(sigstr);
    free(msg);
105 free(buff);
    return msglen;
107 }

109 int reciveServerCiphMsg(Sconexion_t* scnx, char** msg){
    int msglen = 0, bufflen = 0, siglen = 0;
111 char* buff = NULL, *text =  NULL, *sigstr =  NULL;

113 if(scnx == NULL || scnx->socket == -1 || scnx->ctx == NULL || scnx->key == NULL
      ||
      scnx->iv == NULL || scnx->grpkey == NULL || msg == NULL){
115   fprintf(stdout, "Error: NULL input at reciveServerCiphMsg\n");
      return 0;
117 }
    bufflen = recibir(scnx->socket, &buff);
119

    msglen = decrypt_cbc256(scnx->ctx, scnx->key, scnx->iv, (const unsigned char*)
      buff,
121     (unsigned char**)&text, bufflen);
    if(IERROR == strToSigMsgGS(msg, &msglen, &sigstr, &siglen, text, msglen)){
123   fprintf(stdout, "Error: recived invalid message siglen %d msglen %d\n",
      siglen, msglen);
      return 0;
125 }

127 if(!verifySignGS(sigstr, scnx->grpkey, *msg, scnx->scheme))
      return 0;
129

    scnx->iv = realloc(scnx->iv, bufflen);
131 memcpy(scnx->iv, buff, bufflen);
    if(scnx != NULL)
```

```
133     free(scnx->lastsig);
    scnx->lastsig = sigstr;
135   scnx->siglen = siglen;
    free(text);
137   free(buff);
    return msglen;
139 }

141 int sendServerCiphMsg(Sconexion_t* scnx, const unsigned char* text, int textlen){
    int msglen = 0;
143   char*  msg = NULL;
    if(scnx == NULL || scnx->socket == -1 || scnx->ctx == NULL || scnx->key == NULL
       ||
145     scnx->iv == NULL || text == NULL || textlen < 1)
      return 0;
147
    msglen = encrypt_cbc256(scnx->ctx, scnx->key, scnx->iv, text, (unsigned char**)
      &msg, textlen);
149   if(msglen <1)
      return 0;
151   msglen = escribir(scnx->socket, msg, msglen);
    scnx->iv = realloc(scnx->iv, msglen);
153   memcpy(scnx->iv, msg, msglen);

155   free(msg);
    return msglen;
157 }


159
  int reciveClientCiphMsg(Sconexion_t* scnx, char** msg){
161   int bufflen = 0, msglen =0;
    char*  buff = NULL;
163   if(scnx== NULL || scnx->socket == -1 || scnx->ctx == NULL || scnx->key == NULL
      ||
      scnx->iv == NULL || msg == NULL )
165     return 0;
    bufflen = recibir(scnx->socket, &buff);
167   if(bufflen <1)
      return 0;
169   msglen = decrypt_cbc256(scnx->ctx, scnx->key, scnx->iv, (const unsigned char*)
      buff,
      (unsigned char**)msg, bufflen);
171
    scnx->iv = realloc(scnx->iv, bufflen);
173   memcpy(scnx->iv, buff, bufflen);

175   free(buff);
    return msglen;
177 }

179 int revokeClient(Sconexion_t* scnx, groupsig_key_t *mgrkey, gml_t *gml, crl_t *
     crl, char* s_crlf){

181   groupsig_signature_t* sig = NULL;
    message_t* sigmsg = NULL;
183
    sigmsg =  message_init();
185   memcpy(&sigmsg->length, scnx->lastsig, sizeof(uint64_t));
    sigmsg->bytes = malloc(sigmsg->length);
187   memcpy(sigmsg->bytes, scnx->lastsig + sizeof(uint64_t), sigmsg->length);

189   if((sig = groupsig_signature_import(scnx->scheme,
      GROUPSIG_SIGNATURE_FORMAT_MESSAGE_NULL, sigmsg)) ==  NULL){
```

```
    printf("Error: failed to import the signature.\n");
191     return IERROR;
  }
193
  revokeSigGS(sig, scnx->grpkey, mgrkey, gml, crl, scnx->scheme, s_crlf);
195   message_free(sigmsg);
  groupsig_signature_free(sig);
197   return IOK;

199 }


201
int clientInitSConexion(Sconexion_t* scnx){
203   EVP_PKEY* DHkey = NULL, *DHpeerKey = NULL;
  EVP_PKEY_CTX* pctxDH = NULL;
205   char* sigstr = NULL, *msg = NULL, *buff = NULL;
  int siglen = 0, msglen = 0, bufflen = 0;
207
  if(scnx == NULL || scnx->socket < 0 || scnx->keyRSA == NULL || scnx->grpkey ==
    NULL ||
209     scnx->memkey ==  NULL){
    return -1;
211   }
  OpenSSL_add_all_algorithms();
213   groupsig_init(time(NULL));
  /*CHECK RSA MSG*/
215   if(!(bufflen = reciveRSAsign(scnx->socket, scnx->keyRSA, (unsigned char**) &
    buff)))
    return -1;
217
  /*DERIVE AES KEY*/
219   if (!msgToDHpubKey(&DHpeerKey, buff, bufflen)){
    return 0;
221   }

223   if(0 == genKeyFromParamsDH(&pctxDH,&DHkey, DHpeerKey)){
    return 0;
225   }
  free(buff); buff = NULL;
227
  scnx->key = deriveSharedSecretDH(DHkey, DHpeerKey);
229
  /*SEND SIGNED PUBKEY DH*/
231   msglen = DHpubKeyToMsg(DHkey, &msg);
  siglen = signMsgGS(scnx->grpkey, scnx->memkey, scnx->scheme, msg, &sigstr);
233   bufflen = sigMsgToStrGS(msg, msglen, sigstr, siglen, &buff);

235   escribir(scnx->socket, buff, bufflen);

237   free(buff);
  free(msg);
239   free(sigstr);

241   EVP_PKEY_free(DHkey);
  EVP_PKEY_free(DHpeerKey);
243   EVP_PKEY_CTX_free(pctxDH);


245
247   /*CHECK RSA MSG*/
  if(!(bufflen = reciveRSAsign(scnx->socket, scnx->keyRSA, (unsigned char**) &
    buff)))
249     return -1;
```

```
251   /*DERIVE AES IV*/
      if (!msgToDHpubKey(&DHpeerKey, buff, bufflen)){
253     return 0;
      }
255
      if(0 == genKeyFromParamsDH(&pctxDH, &DHkey, DHpeerKey)){
257     return 0;
      }
259   free(buff); buff = NULL;

261   scnx->iv = deriveSharedSecretDH(DHkey, DHpeerKey);

263   /*SEND SIGNED PUBKEY DH*/
      msglen = DHpubKeyToMsg(DHkey, &msg);
265   siglen = signMsgGS(scnx->grpkey, scnx->memkey, scnx->scheme, msg, &sigstr);
      bufflen = sigMsgToStrGS(msg, msglen, sigstr, siglen, &buff);
267
      escribir(scnx->socket, buff, bufflen);
269
      free(buff);
271   free(msg);
      free(sigstr);
273
      EVP_PKEY_free(DHkey);
275   EVP_PKEY_free(DHpeerKey);
      EVP_PKEY_CTX_free(pctxDH);
277
      return 1;
279 }


281
int serverInitSConexion(Sconexion_t* scnx, groupsig_key_t *mgrkey, crl_t* crl,
      gml_t* gml){
283   EVP_PKEY* paramsDH = NULL, *DHkey = NULL, *DHpeerKey = NULL;
      EVP_PKEY_CTX* pctxDH = NULL;
285   char* sigstr = NULL, *msg = NULL, *buff = NULL;
      int siglen = 0, msglen = 0, bufflen = 0;
287
      if(scnx == NULL || scnx->socket < 0 || scnx->keyRSA == NULL || scnx->grpkey ==
        NULL){
289     return -1;
      }
291   OpenSSL_add_all_algorithms();
      /*DH KEY PARAMS FOR AES KEY*/
293   getParamsIniDH(&paramsDH);
      genKeyFromParamsDH(&pctxDH, &DHkey, paramsDH);
295
      /*SIGN RSA MSG*/
297
      msglen = DHpubKeyToMsg(DHkey, &msg);
299   if(sendRSAsign(scnx->socket, scnx->keyRSA, (const unsigned char*)msg, msglen)
        == 0){
        return 0;
301   }
      free(msg);
303   msg = NULL;
      msglen = 0;
305   bufflen = recibir(scnx->socket, &buff);
      if(bufflen < 1)
307     return 0;

309   /*CHECK GS MSG*/
```

```
    strToSigMsgGS(&msg, &msglen, &sigstr, &siglen, buff, bufflen);
311 if(!verifySignGS(sigstr, scnx->grpkey, msg, scnx->scheme))
      return 0;
313
    if(traceSignGS(sigstr, scnx->grpkey, mgrkey, crl, gml, scnx->scheme)){
315   fprintf(stderr, "Clave revocada.\n");
      return 0;
317 }

319 /*DERIVE AES KEY*/
    msgToDHpubKey(&DHpeerKey, msg, msglen);
321 scnx->key = deriveSharedSecretDH(DHkey, DHpeerKey);

323 free(msg); msg = NULL;
    free(sigstr); sigstr = NULL;
325 free(buff); buff = NULL;

327 EVP_PKEY_free(paramsDH); paramsDH = NULL;
    EVP_PKEY_free(DHkey); DHkey = NULL;
329 EVP_PKEY_free(DHpeerKey); DHpeerKey = NULL;
    EVP_PKEY_CTX_free(pctxDH); pctxDH = NULL;
331



333
    /*DH KEY PARAMS FOR AES KEY*/
335 getParamsIniDH(&paramsDH);
    genKeyFromParamsDH(&pctxDH, &DHkey, paramsDH);
337
    /*SIGN RSA MSG*/
339
    msglen = DHpubKeyToMsg(DHkey, &msg);
341 if(sendRSAsign(scnx->socket, scnx->keyRSA, (const unsigned char*)msg, msglen)
      == 0){
      return 0;
343 }
    free(msg);
345 msg = NULL;
    msglen = 0;
347 bufflen = recibir(scnx->socket, &buff);
    if(bufflen < 1)
349   return 0;

351 /*CHECK GS MSG*/
    strToSigMsgGS(&msg, &msglen, &sigstr, &siglen, buff, bufflen);
353 if(!verifySignGS(sigstr, scnx->grpkey, msg, scnx->scheme))
      return 0;
355
    /*DERIVE AES IV*/
357 msgToDHpubKey(&DHpeerKey, msg, msglen);
    scnx->iv = deriveSharedSecretDH(DHkey, DHpeerKey);
359
    free(msg); msg = NULL;
361 free(sigstr); sigstr = NULL;
    free(buff); buff = NULL;
363
    EVP_PKEY_free(paramsDH); paramsDH = NULL;
365 EVP_PKEY_free(DHkey); DHkey = NULL;
    EVP_PKEY_free(DHpeerKey); DHpeerKey = NULL;
367 EVP_PKEY_CTX_free(pctxDH); pctxDH = NULL;

369 return 1;

371 }
```

```
373
int crlRelaod(Sconexion_t* scnx, crl_t** crl, char* s_crl){
375
  crl_free(*crl); *crl = NULL;
377  *crl = crl_import(scnx->scheme, CRL_FILE, s_crl);
  if(*crl == NULL){
379    fprintf(stderr, "Error: invalid crl %s.\n", s_crl);
    return 0;
381  }

383  return 1;

385 }
```

codigos/sconexion.c

N.6. Código del cliente y el servidor

```
1 #include <errno.h>
  #include <sys/types.h>
3 #include <sys/socket.h>
  #include <netinet/in.h>
5 #include <string.h>
  #include <netdb.h>
7 #include <stdio.h>
  #include <stdlib.h>
9 #include <unistd.h>
  #include <time.h>
11 #include <sys/types.h>
  #include <pthread.h>
13 #include <getopt.h>
  #include "../include/comandosu.h"
15 #include "../include/clientui.h"

17 pthread_t h1, h2;
  char* nick;
19 #ifdef TIMETEST
  FILE *f = NULL;
21 #endif

23 Sconexion_t* scnx = NULL;

25 void* hiloEscritura(void* args){
    char* msg = NULL;
27  int end = 0;
    if(scnx == NULL)
29    return NULL;
    while (!end){
31
      msg = scanMsg();
33    if(strlen(msg) == 0){
        free(msg);
35      msg = NULL;
        continue;
37    }
      switch(comando(msg)){
39      case NICK:
          sendNick(scnx, msg, strlen(msg)+1);
41      break;
        case MSG2:
```

```
43          sendMsg(scnx, msg, strlen(msg)+1);
            break;
45          case DISCONNECT:
              end = sendDisconnect(scnx);
47            destroyClientUI();
            break;
49          case PING:
              sendPing(scnx);
51          break;
            case PONG:
53            sendPong(scnx);
            break;
55
            default:
57          break;
        }
59      free(msg);
        msg = NULL;
61    }
      printf("hilo escritura termina\n");
63    pthread_cancel(h2);
      return NULL;
65  }

67
  void* hiloLectura(void* args){
69    char* msg = NULL;
      int end = 0, ret = 0;
71    #ifdef TIMETEST
      clock_t ini, fin, tot;
73    #endif
      if(scnx == NULL)
75      return NULL;
      while(!end){
77      if((ret = reciveClientCiphMsg(scnx, &msg)) <1)
          break;
79
        switch(comando(msg)){
81        case NICK:
            recvNick(msg);
83        break;
          case MSG:
85          recvMsg(msg);
          break;
87        case DISCONNECT:
            recvDisconnect();
89          end = 1;
          break;
91        case PING:
            #ifdef TIMETEST
93          ini = clock();
            recvPing(scnx);
95          fin = clock();
            tot = fin - ini;
97          fprintf(f, "%d\n", tot);
            #else
99          recvPing(scnx);
            #endif
101       break;
          case PONG:
103         recvPong();
          break;
105     }
```

```
    free(msg);
107   }

109   destroyClientUI();
      printf("Se peridio la conexion con el servidor\n");
111   pthread_cancel(h1);
      return NULL;
113 }

115 int identificacion(char* nick){
      return sendNick(scnx, nick, strlen(nick)+1);
117 }

119 int main(int argc , char *argv[]){
      struct addrinfo hints, *res;
121   char opt;
      int long_index=0;
123   char * ip = NULL;
      char * port = NULL;
125   int sockfd = -1;
      void* thstatus;
127
      int scheme = GROUPSIG_CPY06_CODE;
129   groupsig_key_t *grpkey = NULL, *memkey = NULL;
     EVP_PKEY* pubKeyRSA = NULL;
131


133   int key_format;

135   char s_grpkey[] = ".fg/group/grp.key";
      char s_pubKey[] = "pubkey.pub";
137   char* s_memkey = NULL;

139   #ifdef INISCTEST
      clock_t ini, fin, tot;
141   FILE *f;
      f = fopen("iniscnxC.dat", "a");
143   #endif
      static struct option options[] = {
145         {"ip",required_argument,0, 1},
          {"port",required_argument,0, 2},
147         {"nick",required_argument,0, 3},
          {"memkey",required_argument,0, 4},
149
          {0,0,0,0}
151     };
      while ((opt = getopt_long_only(argc, argv,"1:", options, &long_index )) != -1){
153     switch(opt){
        case 1:
155
          ip = malloc(sizeof(char) * strlen(optarg) + 1);
157       strcpy(ip, optarg);
        break;
159     case 2:

161       port = malloc(sizeof(char) * strlen(optarg) + 1);
          strcpy(port, optarg);
163
        break;
165     case 3:
          if (strlen(optarg) > NICK_MAX_LEN){
167         printf("La longuitud del nick debe ser menor de %d caracteres\n",
    NICK_MAX_LEN);
```

```
            return 0;
169         }
            nick = malloc(sizeof(char) * NICK_MAX_LEN + 1);
171         strcpy(nick, optarg);

173       break;
          case 4:
175         s_memkey = malloc(sizeof(char) * strlen(optarg) + 1);
            strcpy(s_memkey, optarg);
177       break;
        }
179   }
    /*Chequeo de que los argumentos esten bien*/
181 if(ip == NULL){
      printf("No se especifico la direccion destino\n");
183   printf("-ip arg -port arg -nick arg -memkey arg\n");
      return 0;
185 }
    if(port == NULL){
187
      printf("No se especifico la puerto destino\n");
189   printf("-ip arg -port arg -nick arg -memkey arg\n");
      return 0;
191 }
    if(nick == NULL){
193
      printf("No se especifico el nick\n");
195   printf("-ip arg -port arg -nick arg -memkey arg\n");
      return 0;
197 }

199 if(s_memkey == NULL){

201   printf("No se especifico la clave de miembro\n");
      printf("-ip arg -port arg -nick arg -memkey arg\n");
203   return 0;
    }
205 memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
207 hints.ai_socktype = SOCK_STREAM;


209
    #ifdef TIMETEST
211 f = fopen("temedioenvioC.dat", "a");
    #endif
213 /*Comenzamos la conexion TCP*/
    if(0!=getaddrinfo(ip, port, &hints, &res)){
215   printf("No se pudo conectar con el servidor\n");
      return 0;
217 }
    sockfd=abrirSocketTCP();
219 if(sockfd==-1){
      return 0;
221 }
    if(-1==abrirConnect(sockfd, *(res->ai_addr))){
223
      return 0;
225 }

227 /*CONEXION SEGURA*/
    switch(scheme) {
229 case GROUPSIG_KTY04_CODE:
      key_format = GROUPSIG_KEY_FORMAT_FILE_NULL_B64;
```

```
231       break;
    case GROUPSIG_BBS04_CODE:
233
    case GROUPSIG_CPY06_CODE:
235
      key_format = GROUPSIG_KEY_FORMAT_FILE_NULL;
237       break;
    default:
239       fprintf(stderr, "Error: unknown scheme.\n");
      return IERROR;
241   }
    groupsig_init(time(NULL));
243   RSAfileToPubKey(&pubKeyRSA, s_pubKey);
    if(pubKeyRSA == NULL){
245       printf("ERR\n");
      return -1;
247   }

249   if(!(grpkey = groupsig_grp_key_import(scheme, key_format, s_grpkey))) {
      fprintf(stderr, "Error: invalid group key %s.\n", s_grpkey);
251       return IERROR;
    }
253
    if(!(memkey = groupsig_mem_key_import(scheme, key_format, s_memkey))) {
255       fprintf(stderr, "Error: invalid member key %s.\n", s_memkey);
      return IERROR;
257   }

259   #ifdef INISCTEST
    ini = clock();
261   #endif

263   scnx = initSconexion(sockfd, grpkey, memkey, scheme, pubKeyRSA);
    if(scnx == NULL || !clientInitSConexion(scnx))
265       return 0;

267   #ifdef INISCTEST
    fin = clock();
269   tot = fin - ini;
    fprintf(f, "%d\n", tot);
271   fclose(f);
    #endif
273
    /*Conexion chat*/
275   identificacion(nick);

277   /*Creacion de la UI*/
    createClientUI();
279
    /*Creacion de los hilos*/
281   pthread_create(&h1,NULL, hiloEscritura, (void *)NULL );
    pthread_create(&h2,NULL, hiloLectura, (void *)NULL );
283   pthread_join(h2, &thstatus);
    pthread_join(h1, &thstatus);
285   close(sockfd);
    free(nick);
287
    #ifdef TIMETEST
289   fclose(f);
    #endif
291
    return 0;
293
```

```
}
```

codigos/cliente.c

```
#include <stdio.h>
#include <string.h> //strlen
#include <arpa/inet.h> //inet_addr
#include <unistd.h> //write
#include <pthread.h>
#include <signal.h>

#include "../include/comandoss.h"

#ifdef TIMETEST
#define PING_SLEEP 10
#define PING_TIME 1
#else

#define PING_SLEEP 30
#define PING_TIME 30

#endif

#define PING_MAX 60
pthread_t* hilos;

LinkedList* listaUsuarios;

EVP_PKEY* privKeyRSA = NULL;
groupsig_key_t *grpkey = NULL;
groupsig_key_t *mgrkey = NULL;
crl_t* crl = NULL;
gml_t* gml = NULL;
char* s_crlf = NULL;
int scheme = GROUPSIG_CPY06_CODE;

#ifdef TIMETEST
FILE* f = NULL;
#endif

void sig_handler(int signo){
  if (signo == SIGINT || signo == SIGTERM){
    #ifdef TIMETEST
    fclose(f);
    #endif
       printf("received end signal\n");
  }
  exit(0);
}


void* controlDeConexion(void* args){
  Node* nd = NULL;
  Usuario_t* usr = NULL;
  unsigned int pingt;
  #ifdef TIMETEST
  clock_t ini, fin, tot;

  f =fopen("tmedioenv.dat", "w");
  #endif
  while(1){
    nd = listaUsuarios->first;
    while (nd != NULL){
```

```
60        printf("PING\n");
          usr = (Usuario_t*) nd->data;
62        pingt = getPingt(usr);
          if((unsigned)time(NULL)- pingt > PING_TIME){
64          #ifdef TIMETEST
            ini = clock();
66          sendPing(usr);
            fin = clock();
68          tot = fin-ini;
            fprintf(f, "%d\n", tot);
70          #else
            sendPing(usr);
72          #endif
          }
74        else if((unsigned)time(NULL) - pingt > PING_MAX){
            sendDisconnect(usr);
76          delete_elem_list(listaUsuarios, (void*) usr);
            liberarUsuario(usr);
78          free(usr);
            usr = NULL;
80
          }
82        nd= nd->next;
        }
84      sleep(PING_SLEEP);
    }
86  }

88
    void* verificarCliente(void* args){
90    int* socket = NULL;
      char * buff = NULL;
92    int bufflen;
      #ifdef INISCTEST
94    clock_t ini, fin, tot;
      FILE *f;
96    f = fopen("iniscnxS.dat", "a");
      #endif
98
      socket = (int*) args;
100   Usuario_t* usr = NULL;
      usr = crearUsuario();
102   if(usr == NULL)
        return NULL;
104
      if(!initUser(usr, *socket, grpkey, NULL, scheme, privKeyRSA)){
106     fprintf(stdout, "Error: failure creating new user.\n");
        liberarUsuario(usr);
108     free(usr);
        usr = NULL;
110     return NULL;
      }
112
      #ifdef INISCTEST
114   ini = clock();
      #endif
116
      if(!serverInitSConexion(usr->scnx, mgrkey, crl, gml)){
118     fprintf(stdout, "Error: failure creating secure conexion.\n");
        liberarUsuario(usr);
120     free(usr);
        usr = NULL;
122     return NULL;
```

```
  }
124
126 #ifdef INISCTEST
  fin = clock();
128 tot = fin - ini;
  fprintf(f, "%d\n", tot);
130 fclose(f);
  #endif
132
  while(1){
134
    bufflen = reciveServerCiphMsg(usr->scnx, &buff);
136 if(bufflen < 1){
      fprintf(stdout, "Error: invalid message at identification\n");
138   delete_elem_list(listaUsuarios, (void*) usr);
      liberarUsuario(usr);
140   free(usr);
      usr = NULL;
142   return NULL;
    }
144
    if(comando(buff)== NICK){
146   recvNick(usr, buff);
      fprintf(stdout, "Se registro %s\n", usr->nick);
148   sendServerCiphMsg(usr->scnx, (const unsigned char*)"/MSG server bienvenido"
    ,
                strlen((char*)"/MSG server bienvenido")+1);
150   //escribir(*usr->socket, "/MSG server bienvenido", strlen((char*)"/MSG
    server bienvenido")+1);
      break;
152 }
    else{
154   sendServerCiphMsg(usr->scnx, (const unsigned char*)CDISCONNECT,
              strlen((char*)CDISCONNECT)+1);
156   //escribir(*usr->socket, CDISCONNECT, strlen((char*)CDISCONNECT)+1);

158 }

160 }
  insert_list(listaUsuarios, usr);
162 return usr;
}
164

166 void* lecturaUsuario(void* args){
  char * buff = NULL;
168 int bufflen, end = 0;
  Usuario_t* usr = verificarCliente(args);
170 if(usr == NULL){
    fprintf(stdout, "Error: fallo al verificar cliente\n");
172   return NULL;
  }
174 while(!end){
    bufflen = reciveServerCiphMsg(usr->scnx, &buff);
176   if(bufflen < 1){
      fprintf(stdout, "%d\n", delete_elem_list(listaUsuarios, (void*) usr));
178   recvDisconnect(usr);
      liberarUsuario(usr);
180   free(usr);
      usr = NULL;
182   end = 1;
      fprintf(stdout, "cosas extrañas pueden pasar\n");
```

```
184       break;
      }
186     fwrite(buff, 1, bufflen, stdout);
      fprintf(stdout, "bufflen: [%d]\n", bufflen);
188     switch (comando(buff)){
        case NICK:
190       fprintf(stdout, "NICK\n");
          recvNick(usr, buff);
192       break;

194       case MSG:
          fprintf(stdout, "MSG\n");
196       if(recvMsg(buff, bufflen)){
            /*SE REVOCA AL MIEMBRO*/
198         revokeClient(usr->scnx, mgrkey, gml, crl, s_crlf);
            printf("se empieza a recargar el crl\n");
200         crlRelaod(usr->scnx, &crl, s_crlf);
            printf("se recargo el crl\n");
202
            recvDisconnect(usr);
204         liberarUsuario(usr);
            free(usr);
206         end = 1;
          }
208       break;

210       case DISCONNECT:
          fprintf(stdout, "DISCONNECT\n");
212       recvDisconnect(usr);
          liberarUsuario(usr);
214       free(usr);
          end = 1;
216       break;

218       case PING:
          fprintf(stdout, "PING\n");
220       recvPing(usr);
          break;
222
          case PONG:
224       fprintf(stdout, "PONG\n");
          recvPong(usr);
226       break;

228       default:
          fprintf(stdout, "default\n");
230
          break;
232



234
      }
236     free(buff);
    }
238   fprintf(stdout, "Usuario desconectado\n");

240   return NULL;
}
242


244 int main(){
    int socket;
246   int* socketcli;
```

```
  int usuarios = 0;
248 pthread_t hiloPing;
  struct sockaddr_in ip4addr;
250
  char s_grpkey[] = ".fg/group/grp.key";
252 char s_crl[] = ".fg/manager/crl";
  char s_gml[] = ".fg/manager/gml";
254 char s_mgrkey[] = ".fg/manager/mgr.key";

256 s_crlf = s_crl;

258 char s_rsaPrivKey[] = "privkey.pem";

260 RSAfileToPrivKey(&privKeyRSA, s_rsaPrivKey);
  if(privKeyRSA == NULL){
262   fprintf(stdout, "Error: failure loading RSA key\n");
    return 0;
264 }

266 /*CONFIGURACION DE SEÑAELS DEL SISTEMA*/

268 if (signal(SIGINT, sig_handler) == SIG_ERR)
    printf("\ncan't catch SIGINT\n");
270 if(signal(SIGTERM, sig_handler) == SIG_ERR)
    printf("\ncan't catch SIGTERM\n");
272 /*INICIAIZACION DE COSAS VARIAS*/
  iniBigBrother(NULL);
274 groupsig_init(time(NULL));

276 if(0 == import_manager(&grpkey, &mgrkey, &crl, &gml,
      s_grpkey, s_mgrkey, s_crl, s_gml, scheme))
278   return 0;

280 socket = abrirSocketTCP();
  if(socket < 1)
282   return 0;

284 if(abrirBind(socket, 8080) < 0)
    return 0;
286
  if(abrirListen(socket) == -1)
288   return 0;


290
  listaUsuarios = create_list(compareUsr);
292 pthread_create(&hiloPing,NULL, controlDeConexion, NULL );

294 while(1){
    socketcli = malloc(sizeof(int));
296   *socketcli = aceptar(socket, ip4addr);

298   fprintf(stdout, "nuevo cliente\n");
    usuarios++;
300   hilos= realloc(hilos, sizeof(pthread_t)*usuarios);

302   pthread_create(&hilos[usuarios-1],NULL, lecturaUsuario, (void*) socketcli);
    pthread_detach(hilos[usuarios-1]);
304
  }
306
  fprintf(stdout, "fin\n");
308 free(privKeyRSA);
  free(grpkey);
```

```
310   free(mgrkey);
      free(crl);
312   free(gml);
      freeBigBrother();
314
      return 0;
316 }
```

codigos/server.c

```
1  #include "../include/usuario.h"

3  int nUsr = 0;

5  Usuario_t* crearUsuario(){
     Usuario_t* usr = NULL;
7
     usr = malloc(sizeof(Usuario_t));
9    usr->id = nUsr++;
     usr->nick = malloc(NICK_MAX_LEN + 1);
11   usr->nick[0] = '\0';
     setCurrentPingt(usr);
13   return usr;
   }
15
   int initUser(Usuario_t* usr, int socket, groupsig_key_t *grpkey, groupsig_key_t *
       memkey, int scheme, EVP_PKEY* keyRSA){
17   if(usr == NULL)
       return 0;
19   usr->scnx = initSconexion(socket, grpkey, memkey, scheme, keyRSA);
     if(usr->scnx == NULL)
21     return 0;
     return 1;
23 }

25 int liberarUsuario(Usuario_t* usr){
     if(usr == NULL)
27     return 1;
     if(!(usr->nick == NULL)){
29     free(usr->nick);
       usr->nick = NULL;
31   }
     freeSconexion(usr->scnx);
33   free(usr->scnx); usr->scnx = NULL;
     return 0;
35 }

37 int setNick(Usuario_t* usr ,char* nick){
     int len = 0;
39   if(nick == NULL)
       return -1;
41
     if(usr->nick == NULL)
43     return -2;

45   len = strlen(nick);

47   if (len > NICK_MAX_LEN || len ==0)
       return 1;
49
     strcpy(usr->nick, nick);
51   return 0;
   }
```

```
53
char* getNick(Usuario_t* usr){
55   if (usr == NULL)
       return NULL;
57   return usr->nick;
}
59
int setId(Usuario_t* usr, int id){
61   if(usr == NULL)
       return -1;
63   usr->id = id;
     return 0;
65
}
67 int getId(Usuario_t* usr){
     if(usr == NULL)
69     return -1;
     return usr->id;
71
}
73

75 int setCurrentPingt(Usuario_t* usr){
     if(usr == NULL)
77     return -1;
     usr->pingt= (int) time(NULL);
79   return 0;
}
81
int getPingt(Usuario_t* usr){
83   if(usr == NULL)
       return -1;
85   return usr->pingt;
}
87

89 int setKey(Usuario_t* usr, char** key){
     if(usr == NULL || key == NULL || *key == NULL || usr->scnx == NULL)
91     return 0;

93   usr->scnx->key = (unsigned char*)*key;
     return 1;
95 }
char* getKey(Usuario_t* usr){
97   if(usr == NULL)
       return NULL;
99
     return (char*)usr->scnx->key;
101 }

103 int setIv(Usuario_t* usr, char** iv){
     if(usr == NULL || iv == NULL || *iv == NULL || usr->scnx == NULL)
105    return 0;

107  usr->scnx->iv = (unsigned char*)*iv;
     return 1;
109 }
char* getIv(Usuario_t* usr){
111  if(usr == NULL)
       return NULL;
113
     return (char*)usr->scnx->iv;
115 }
```

```
117
int compareUsr(const void* data1, const void* data2){
119   Usuario_t* usr1, * usr2;
   if(data1 == NULL || data2 == NULL)
121     return -1;

123   usr1= (Usuario_t*) data1;
   usr2= (Usuario_t*) data2;
125   if (usr1->id == usr2->id)
     return 0;
127   else
     return 1;
129 }
```

codigos/usuario.c

```
1 #ifndef USUARIO_H
  #define USUARIO_H
3
  #include <stdio.h>
5 #include <stdlib.h>
  #include <string.h>
7 #include <time.h>
  #include "sconexion.h"
9
  #define NICK_MAX_LEN 8
11
  /*Por ahora esto, pero es posible que haya que añadir cosas de claves*/
13 typedef struct USUARIO
  {
15   int id;
   int pingt;
17   char* nick;
   Sconexion_t * scnx;
19 }Usuario_t;


21
  Usuario_t* crearUsuario();
23 int initUser(Usuario_t* usr, int socket, groupsig_key_t *grpkey, groupsig_key_t *
     memkey, int scheme, EVP_PKEY* keyRSA);

25 int liberarUsuario(Usuario_t* usr);

27 int setNick(Usuario_t* usr, char* nick);
  char* getNick(Usuario_t* usr);
29



31
  int setId(Usuario_t* usr, int id);
33 int getId(Usuario_t* usr);


35
  int setKey(Usuario_t* usr, char** key);
37 char* getKey(Usuario_t* usr);

39 int setIv(Usuario_t* usr, char** iv);
  char* getIv(Usuario_t* usr);
41
  int setCurrentPingt(Usuario_t* usr);
43 int getPingt(Usuario_t* usr);

45 int compareUsr(const void* usr1, const void* usr2);
```

```
47

49 #endif
```

codigos/usuario.h

N.7. Código de la interfaz de usuario

```
1  #include "../include/clientui.h"
3
   int pos = 0, row, col, max;
5
   WINDOW* win1, * win2, * win3, * win4;
7
   void getWIndowSize(){
9    getmaxyx(stdscr,row,col);
     max = row - 8;
11 }
13 WINDOW *createNewWin(int height, int width, int starty, int startx){  WINDOW *
       local_win;
15   local_win = newwin(height, width, starty, startx);
     box(local_win, 0 , 0);     /* 0, 0 gives default characters
17             * for the vertical and horizontal
               * lines       */
19   wrefresh(local_win);     /* Show that box     */
21   return local_win;
   }
23 WINDOW *createNewWinHidden(int height, int width, int starty, int startx){
       WINDOW *local_win;
25   local_win = newwin(height, width, starty, startx);
     box(local_win, 0 , 0);     /* 0, 0 gives default characters
27             * for the vertical and horizontal
               * lines       */
29   wborder(local_win, ' ', ' ', ' ',' ',' ',' ',' ',' ',' ');
     wrefresh(local_win);     /* Show that box     */
31
     return local_win;
33 }
35 void destroyWin(WINDOW *local_win){
     /* box(local_win, ' ', ' '); : This won't produce the desired
37    * result of erasing the window. It will leave it's four corners
      * and so an ugly remnant of window.
39    */
     wborder(local_win, ' ', ' ', ' ',' ',' ',' ',' ',' ',' ');
41   /* The parameters taken are
      * 1. win: the window on which to operate
43    * 2. ls: character to be used for the left side of the window
      * 3. rs: character to be used for the right side of the window
45    * 4. ts: character to be used for the top side of the window
      * 5. bs: character to be used for the bottom side of the window
47    * 6. tl: character to be used for the top left corner of the window
      * 7. tr: character to be used for the top right corner of the window
49    * 8. bl: character to be used for the bottom left corner of the window
      * 9. br: character to be used for the bottom right corner of the window
```

```
51     */
   wrefresh(local_win);
53   delwin(local_win);
}
55

57 void printMsg(char* msg){
   char* str = malloc(sizeof(char)* (strlen(msg) + 2));
59   sprintf(str, "%s\n", msg);
   mvwaddstr(win3, pos, 0, str);
61   wrefresh(win3);
   pos++;
63   pos += (strlen(str)-1)/(col -2);
   pos= pos%max;
65 }
char* scanMsg(){
67   char* str;
   str = malloc(BUFF_LEN);
69   mvwgetnstr(win4, 0, 0, str, BUFF_LEN);
   str = realloc(str, strlen(str)+1);
71   destroyWin(win4);
   win4 = createNewWinHidden(4, col -2 , row - 5, 1);
73   return str;
}
75
void createClientUI(){
77   initscr();
   getWIndowSize();
79   win1 = createNewWin(row - 6, col, 0, 0);
   win2 = createNewWin(6, col, row - 6, 0);
81   win3 = createNewWinHidden(row - 8, col - 2, 1, 1);
   win4 = createNewWinHidden(4, col -2 , row - 5, 1);
83 }

85 void destroyClientUI(){
   destroyWin(win1);
87   destroyWin(win2);
   destroyWin(win3);
89   destroyWin(win4);
   endwin();
91 }

93



95
/*int main(int argc, char** argv){
97   int row, col;
   WINDOW* win1, * win2, * win3, * win4;
99   int pos = 0, max;

101   char* str = malloc(sizeof(char)* 8096);

103   initscr();
   getmaxyx(stdscr,row,col);
105   max = row - 8;
   win1 = create_newwin(row - 6, col, 0, 0);
107   win2 = create_newwin(6, col, row - 6, 0);
   win3 = create_newwin_hidden(row - 8, col - 2, 1, 1);
109   win4 = create_newwin_hidden(4, col -2 , row - 5, 1);


111
   while(1){
113     mvwgetstr( win4, 0, 0, str);
```

```
    destroy_win(win4);
    win4 = create_newwin_hidden(4, col -2 , row - 5, 1);
    sprintf(str, "%s\n", str);
    mvwaddstr(win3, pos, 0, str);
    wrefresh(win3);
    pos++;
    pos += (strlen(str)-1)/(col -2);
    pos= pos%max;
  }
  destroy_win(win1);
  destroy_win(win2);
  destroy_win(win3);
  destroy_win(win4);
  endwin();
  return 0;
}*/
```

codigos/clientui.c

N.8. Código de la revocación

```
#include "../include/parser.h"

char** banWords = NULL;
int nwords;

int parseNick(char* src, char** nick1, char ** nick2){
  char *pch1 = NULL, *pch2 = NULL;
  *nick1 =  NULL;
  *nick2 =  NULL;

  if(src == NULL || strlen(src) < 1 || nick1 == NULL || nick2 == NULL){
    return -1;
  }
  if((pch1 = strchr(src, ' ')) == NULL)
    return -1;
  pch1++;

  if((pch2 = strchr(pch1, ' ')) == NULL){
    *nick1 = malloc(strlen(pch1) + 1);
    memcpy(*nick1, pch1, strlen(pch1) + 1);
  }
  else{
    *nick1 = malloc(pch2 - pch1 +1);
    memcpy(*nick1, pch1, pch2 - pch1);
    (*nick1)[pch2 - pch1] = '\0';
  }
  if(pch2 == NULL){
    return 0;
  }
  pch2 ++;
  *nick2 = malloc(strlen(pch2) + 1);
  memcpy(*nick2, pch2, strlen(pch2) + 1);
  return 0;
}

int parseMsg(char* src, char** nick, char ** msg){
  int msglen = 0;
  int nlen = 0;
  char *pch = NULL;
  char *pch2 = NULL;
```

```
41
   if(src == NULL || strlen(src) < 1 || nick == NULL || msg == NULL){
43   return 0;
   }
45 if((pch = strchr(src, ' ')) == NULL)
     return 0;
47 pch++;
   if((pch2 = strchr(pch, ' ')) == NULL)
49   return 0;

51 nlen = pch2 - pch + 1;
   *nick =  malloc(nlen);
53 memcpy(*nick, pch, nlen-1);
   (*nick)[nlen-1] = '\0';
55
   pch2++;
57 msglen = strlen(pch2)+1;
   *msg = malloc(msglen);
59 memcpy(*msg, pch2, msglen-1);
   (*msg)[msglen-1] = '\0';
61
   return 1;
63 }
   void iniBigBrother(char* fname){
65
   /*file* f = NULL;
67 if(fname !=NULL){

69 }*/
   banWords =  malloc(sizeof(char*));
71 banWords[0] = malloc(strlen("cara anchoa")+1);
   strcpy(banWords[0], "cara anchoa");
73 nwords = 1;
   }
75
   int bigBrother(char* src){
77   int i = 0, msglen = 0, ret = 0;
     char *msg = NULL, *nick = NULL;
79


81   if(src == NULL || strlen(src)<1)
       return 0;
83   printf("[%s]\n", src);
     parseMsg(src, &nick, &msg);
85
     msglen = strlen(msg)+1;
87   for (; i < nwords; i++){
       if(msglen < strlen(banWords[i]))
89       continue;

91     if(strstr(msg, banWords[i]) != NULL){
         ret = 1;
93       break;
       }
95   }
     free(nick);
97   free(msg);
     return ret;
99 }

101 void freeBigBrother(){
     int i = 0;
103  for (; i < nwords; i++){
```

```
        free(banWords[i]);
    }
    free(banWords);
}
```

codigos/parser.c